Cross-platform Open Security Stack for Connected Device

D4.3 CROSSCON Extensions to Domain Specific Hardware Architectures Documentation - Final

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

Related WP	WP4	**Document Reference**	D4.3
Related Deliverable(s)	D1.4, D1.5, D2.1 D4.1, D4.2	**Dissemination Level(*)**	PU
Lead Participant	UWU	**Lead Author**	Melanie Götz (UWU)
Contributors	BEYOND, TUD	**Reviewers**	Ainara García (BIOT), Hristo Koshutanski (ATOS)

Keywords
HW-SW codesign, TEE, CROSSCON SoC, BA51-H, FPGA, FPGA TEE, vFPGA, Perimeter guard

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

Document Information

List of contributors	
Name	**Partner**
Melanie Götz	UWU
Tymoteusz Burak	UWU
Fabian Schmitt	UWU
Alberto Tacchella	UNITN
Michele Grisafi	UNITN
Carlo Ramponi	UNITN
Nikhilesh Singh	TUD
Ziga Putrle	BEYOND

Document history			
Ver.	**Date**	**Change editors**	**Changes**
0.1	22/05/2025	Melanie Götz (UWU)	First draft - Initial Contributions
0.2	30/07/2025	Nikhilesh Singh (TUD)	Added TUD-relevant portion - first draft
0.3	31/07/2025	Nikhilesh Singh (TUD)	Added TUD-relevant portion - second draft
0.4	01/08/2025	Žiga Putrle (BEYOND)	Adding missing content for BA51-H, PG and AES-GCM accelerator sections.
0.5	01/08/2025	Tymoteusz Burak (UWU)	Adding missing content for Physical Unclonable Function Hardware Primitive.
0.6	01/08/2025	Fabian Schmitt (UWU)	Adding missing content for Environmental Fingerprinting Hardware Primitive.
0.7	04/08/2025	Nikhilesh Singh (TUD)	Added TUD-relevant portion - final draft
0.8	22/08/2025	Žiga Putrle (BEYOND), Hamid Dashtbani (UWU), Nikhilesh Singh (TUD)	Addressing comments from the reviewers.
0.9	29/08/2025	Juan Alonso	Quality Assessment.
1.0	29/08/2025	Hristo Koshutanski	Final version submitted.

Quality Control		
Role	**Who (Partner short name)**	**Approval Date**
Deliverable leader	Melanie Götz (UWU)	25/08/2025
Quality Manager	Juan Alonso (ATOS)	29/08/2025
Project Coordinator	Hristo Koshutanski (ATOS)	29/08/2025

Table of Contents

List of Tables

List of Figures

List of Abbreviations

Abbreviation / acronym	Description
ABI	Application Binary Interface
AES	Advanced Encryption Standard
AES-256-GCM	Advanced Encryption Standard with Galois/Counter Mode 256-bits
AES-GCM	Advanced Encryption Standard with Galois/Counter Mode
AIA	Advanced Interrupt Architecture
API	Application Programming Interface
APU	Application Processing Unit
ASLR	Address Space Layout Randomization
BOM	Bill of Materials
BRAM	Block Random Access Memory
CA	Client Application
CA1	Client1's Application
CA2	Client2's Application
CAs	Client Applications
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
DSP	Digital Signal Processor
EL2	Exception Level 3
EL3	Exception Level 3
FOSE	Firmware Object Signing and Encryption
FOTA	Firmware OTA
FPGA	Field-Programmable Gate Array
FPGAs	Field-Programmable Gate Arrays
GCM	Galois counter mode
GIC	General Interrupt Controller
GP	GlobalPlatform
GPOS	General Purpose OS
GPU	Graphic Processing Unit
HAL	Hardware Abstraction Layer
HBOM	Hardware BOM
HW	Hardware
HW IP	Hardware IP
Hypv.	CROSSCON Hypervisor

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
LUTs	Look-Up Table
MAC	Media Access Control
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
NS	Non-Secure
OS	Operating Systems
OTA	Over-the-Air
OTP	One-Time-Programmable
PG	Perimeter guard
PKCS	Public-Key Cryptography Standards
PL	Programmable Logic
PLIC	Platform Local Interrupt Controller
PMP	Physical Memory Protection
PMU	Performance Monitor Unit
PMU FW	PMU Firmware
PR	Partial Reconfiguration
PS	Processing System
pTA	pseudo Trusted Application
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
RSA-PKCS	Rivest-Shamir-Adleman Public-Key Cryptography Standards
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
SHA	Secure Hash Algorithm
SIEM	Security Information and Event Management
SiP	Silicon Provider
SMC	Secure Monitor Call
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
STM	Satisfiability Modulo Theories
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
TF-A	ARM Trusted Firmware
TinyML	Tiny Machine Learning
TPM	Trusted Platform Module
TUF	The Update Framework
UC	Use Case
UUID	Universally Unique Identifier
vFPGA	virtual FPGA
VM	Virtual Machine
VMM	VM Monitor
Wi-Fi	Wireless Fidelity
WP	Work Package

Executive Summary

When considering the security of IoT (Internet of Things) devices, we also need to take into consideration the underlying system on chip (SoC) on which our software (SW) is running on. Usually, IoT devices are developed by using off-the-shelf SoCs, which are an adequate fit for many use cases. But if we want to have greater control, transparency, and flexibility of how our IoT devices behave and improve their security, we need to design our own SoCs. As part of the CROSSCON project, we are researching and designing new hardware-software features that improve security of IoT devices, especially when used together with the CROSSCON Stack. We mainly focus on security of domain specific hardware, which is often overlooked in existing SoCs.

In this document, we present the final results of the work done as part of WP4 that can be divided into five related areas: (i) a custom RISC-V based SoC, called CROSSCON SoC, that can be used together with the CROSSCON Stack to provide strong security guarantees; (ii) a unified interface that allows trusted services to access domain specific hardware, such us cryptographic accelerators, Physical Unclonable Function (PUF) or Wireless Fidelity (Wi-Fi) data for environmental fingerprinting; (iii) a mechanism that allows sharing of hardware (HW) accelerators between isolated domains without compromising the isolation between the domains, called Perimeter Guard; (iv) a TEE-like environment on field-programmable gate arrays (FPGAs) that allows users to extend their trusted applications with trusted FPGA-accelerated tasks; (v) a runtime exploit mitigation mechanism that leverages the ARM Memory Tagging Extensions (MTE) to provide a persistent and deterministic defense against stack-based spatial memory errors. By leveraging any of the aforementioned solutions together with the CROSSCON Stack, one can improve the security of their IoT device.

This deliverable D4.3 contributes to the accomplishment of milestone MS9, 'Final version of integrated CROSSCON stack and extension primitives.' It presents the cumulative effort undertaken within WP4.

1 Introduction

1.1 Purpose of the Document

This document presents the research and development outcomes of WP4, which focuses on strengthening the security and adaptability of IoT devices. The document describes the design and implementation of custom CROSSCON System-on-Chips (SoCs), developed to offer enhanced control, transparency, and security compared to commercial off-the-shelf alternatives.

The document also introduces interface definitions for hardware primitives of the CROSSCON Stack including cryptographic accelerator, PUF and environmental fingerprinting. Those were designed to streamline the development of secure IoT applications across heterogeneous platforms.

Additionally, it describes Perimeter Guard, a mechanism that enables secure sharing of hardware components, such as cryptographic accelerators, among isolated domains without compromising their separation. Furthermore, it discusses the optimization of FPGA usage to support multiple hardware accelerators, outlining the techniques employed to ensure robust domain isolation and secure reconfiguration.

Finally, it describes a runtime exploit mitigation mechanism that leverages the ARM Memory Tagging Extensions to provide a persistent and deterministic defense against stack-based spatial memory errors.

1.2 Relation to Other Project Work

The work described in this document is closely related to work done in WP2, WP3, and WP5. The WP2 provided the initial research results and requirements needed for the development, where WP3 was co-developed with WP4 as the solutions need to be compatible and complement each other. Furthermore, the work in WP4 was used to refine and inform the development of open specification and formal framework in WP2, and was used as part of WP5 to further integrated and tests the solutions. The research and development work in WP4 aligns with the overarching project objectives, specifically focusing on enhancing hardware-level security within the CROSSCON Stack.

This document contributes to the accomplishment of milestone MS8 “Final version of the CROSSCON stack components and extension primitives, and extended testbed” and MS9 "Final version of integrated CROSSCON stack, and extension primitives", and it’s closely related to the deliverable D4.4 “CROSSCON Extension Primitives to Domain Specific Hardware Architectures - Final Version”, as it contains the implementation results described in this document.

1.3 Structure of the Document

The document is structured into several sections that describe different parts of our work. The document begins with an overview section, Chapter 2, which provides a short synopsis of the four following sections. Following this, Chapter 3 discusses the aforementioned interfaces for hardware primitives. Then, Chapter 4 introduces the CROSSCON SoC, a custom SoC that can be used together with the CROSSCON Stack to improve the security of IoT devices, as well as Perimeter guard, AES-GCM accelerator with context switching support, and TAGShield, a runtime exploit mitigation mechanism that prevents stack-based spatial memory errors. Subsequently, Chapter 5 discusses securely provisioning FPGAs through

providing a TEE-like environment that can be used for FPGA-based acceleration while protecting the Hardware IP of clients. Finally, Chapter 6 concludes the document by summarizing key findings.

1.3.1 Changes w.r.t. D4.1

This document extends the draft presented in D4.1. Chapter 3 describes two new hardware primitive interfaces: PUF and Wi-Fi based environmental fingerprinting. Chapter 4 has been adapted and extended to emphasis how the cross hardware-software stack isolation works. Providing additional details about the architecture and implementation of the CROSSCON SoC, Perimeter guard and context switching support for AES-GCM accelerator. Furthermore, a new section was added to describe TAGShield, a runtime exploit mitigation mechanism that prevents stack-based spatial memory errors. Note that some material was also moved to other deliverables, for example, description of Perimeter guard, as the description was more sutable for related deliverables. Chapter 5 presents the end-to-end final architecture and implementation of Secure FPGA Provisioning integrated with the CROSSCON Hypervisor. Further, it also presents the details of the associated demonstrator. In contrast, the corresponding section in D4.1 discussed the initial bare-metal implementation with a single client.

2 Overview

The goal of WP4 is to improve the security of IoT devices by co-designing new HW/SW features that complement the CROSSCON Stack. We mainly focus on providing additional security for domain-specific hardware and architecture, as, for example, HW accelerators, FPGAs, and peripheral devices. Our work can be divided into five related areas that can be put into a context of an hardware-software stack as shown in Figure 1. We discuss each area in detail in the following sections.

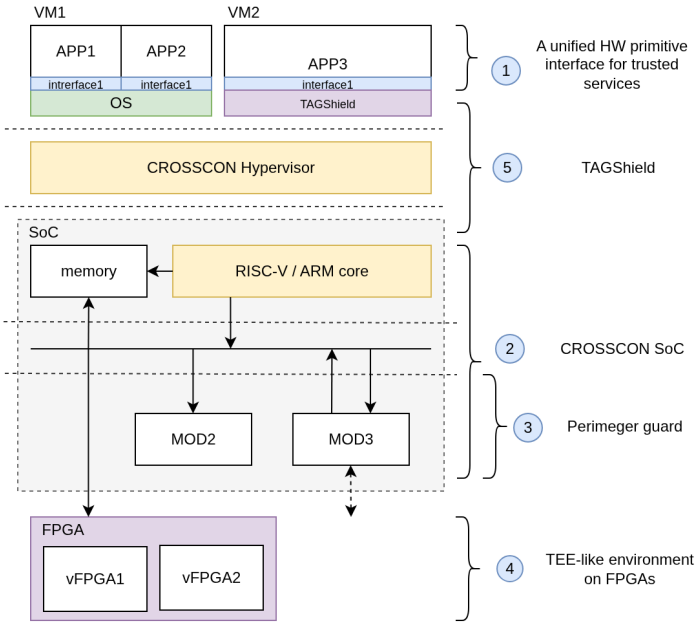


Figure 1: How different areas of our WP4 fit into a hardware-software stack

Figure 1 shows a simplified hardware-software stack that demonstrates how the main five areas of our work are related: (1) a unified interface that allows Trusted services to access domain specific hardware, (2) CROSSCON SoC, (3) set of HW features that enable sharing of HW accelerators without compromising isolation between domains, called Perimeter guard, and (4) a mechanism that allows FPGA provisioning and enables TEE-like environment on FPGA.

A unified interface for trusted services (1): To simplify the development processes of secure IoT devices, the CROSSCON Stack needs to provide a common interface through which cryptographic primitives can be used across different platforms. The main role of the interface is to abstract away the platform-specific details (e.g., how the underlying cryptographic HW Accelerator is used) and provide a common interface that is easy to use and has a clear description of functionality that it provides. While numerous vendors deploy their own proprietary APIs for domain-specific accelerators, there have been some efforts from both industry and academia to specify a unified API. These efforts have led to various proposals tailored to different device classes and use cases. We investigate the two most popular APIs: the PSA Crypto API and the GlobalPlatform (GP) Cryptographic Operations API.

We also provide custom interfaces that abstract the hardware primitives used by the trusted services of the CROSSCON stack, particularly Physical Unclonable Functions (PUFs) and Wi-Fi chips used for environmental fingerprinting, as a proposed step toward their unification and standardization across the industry.

CROSSCON SoC (2): The security of IoT devices also depends on the security of the underlying SoC on which the software is running on. Usually, IoT devices are developed by using off-the-shelf SoCs, which are an adequate fit for many use cases. But if we want to have greater control, transparency, and flexibility of how our IoT devices behave and improve their security, we need to design our own SoCs. As part of WP4, we designed a custom SoC, called CROSSCON SoC, that can be used together with the CROSSCON Stack to improve the security of IoT devices. The CROSSCON SoC includes a BA51-H (RISC-V) core that supports a novel TEE design that leverages hardware virtualization primitives to isolate software (e.g., RTOS and applications) into dedicated virtual execution environments (VMs).

Perimeter guard (3): To share a HW module (for example, a cryptographic HW accelerator) across different isolated domains while preserving the isolation, we need to make sure that there are no unwanted information flows going through the HW module from one domain to another. However, accounting for all possible information flows passing through a specific module is often hard and time consuming. Because of that, it is usually not done in the industry. This leads to a situation where a HW module is limited to a single domain (the safe way) or it is shared between domains, which can compromise the existing isolation (the unsafe way). As part of WP4, we looked into solutions that are easy to apply and would allow us to share a module across domains while preserving the isolation guarantees. We present an HW module, called Perimeter guard (PG), as a promising solution that can address this problem.

TEE-like environment on FPGAs (4): FPGAs are configurable hardware platforms that can be configured by end users to implement any digital circuit, in this context, any HW accelerator. The FPGA's programmable fabric constitutes a single physical unit that can be programmed or configured at once. However, with the enhanced capabilities of modern FPGAs, it is feasible to accommodate multiple hardware accelerators simultaneously. As part of WP4, we optimized FPGA utilization while ensuring strict isolation between different domains that share the FPGA fabric simultaneously, guaranteed by the CROSSCON Hypervisor.

TAGShield (5): Stack memory safety violations, such as buffer overflows, continue to pose a significant security challenge in software developed using memory-unsafe languages such as C and C++. Hardware advancements such as memory tagging schemes offer a promising foundation for enhancing memory safety by enabling the tagging of both pointers and memory objects. Unfortunately, existing tag-based approaches fail to protect pointer tags, which can be manipulated during pointer arithmetic operations that are common in system-level code, thereby rendering the security guarantees provided by current solutions questionable. As part of WP4, we present TAGShield: a runtime exploit mitigation mechanism that leverages the ARM Memory Tagging Extensions (MTE) to provide a persistent and deterministic defense against stack-based spatial memory errors, while preserving the integrity of pointer tags. TAGShield provides strong security guarantees through a transparent compile-time tagging scheme, complemented by a lightweight software instrumentation layer that enforces tag integrity.

All five areas of our work address different problems that we encounter when designing an IoT system. However, the suggested solutions complement each other across the hardware-software stack (Figure 1). For example, we can use CROSSCON SoC together with the CROSSCON Hypervisor to establish isolated virtual machines (VMs). We could then use PGs in CROSSCON SoC to share HW accelerators across VMs while preserving isolation between them. Furthermore, we could use the TEE-like environment on the FPGA to provide several different HW accelerators that use the same FPGA fabric with strong isolation between them. Additionally, the software running in the VMs can use the available HW modules through the unified interface that hides the details of how the accelerators are used. And finaly, if CROSSCON SoC would also include an ARM core, we could use TAGShield to provide a persistent and deterministic defense against stack-based spatial memory errors by leveraging ARM's MTE.

In the rest of the document, we describe each area of our work in detail.

3 Extension Primitives for Trusted Services

To support the trusted services of the CROSSCON stack, both the existing ones and the new trusted services developed as part of WP3, a set of extension primitives was provided for use by these services. As part of CROSSCON's objective to support trusted services across heterogeneous devices, we have identified and implemented three critical hardware interfaces that serve as primitives for such services: *Cryptographic Accelerators*, *physically unclonable Functions (PUFs)*, and *Hardware primitives for Environmental Fingerprinting*.

These interfaces reflect a broader design goal: enabling interoperability and secure abstraction over diverse hardware capabilities. Cryptographic accelerators provide hardware-level support for essential operations like encryption, hashing, and digital signatures. PUFs offer a physically anchored source of device uniqueness, enabling secure key generation and device authentication without requiring stored secrets. Finally, environmental fingerprinting through Wi-Fi Channel State Information (CSI) captures the physical characteristics of wireless transmissions to provide context-aware security, such as location-bound verification.

In this section, we present unified interfaces for these hardware primitives, enabling the trusted services of the CROSSCON stack to utilize these device-specific capabilities when available, while remaining agnostic of the underlying platform. This chapter introduces each of these three interfaces in detail and outlines their current implementations.

3.1 Overview of cryptographic accelerators and their current presence in the market

To establish a unified interface that facilitates the integration of domain-specific hardware across a diverse array of heterogeneous devices, it is crucial to first examine the current landscape of IoT devices. Our initial effort focused on analyzing the domain-specific hardware currently employed in IoT devices. Specifically, we prioritized understanding the state of cryptographic accelerators, given their direct relevance to security and their presence in a variety of IoT devices, spanning both low and high-end spectrums. While the popularity of other domain-specific hardware, like machine learning accelerators, continues to rise, their availability remains predominantly confined to higher-end IoT devices. Consequently, this section will detail our findings regarding the cryptographic accelerators in commercially available IoT devices, highlighting their capabilities and interfaces.

ARM CryptoCell: The ARM CryptoCell-300 series [1] embodies an exhaustive array of security solutions tailor-made for embedded systems. This series serves as a robust security subsystem, featuring accelerators for public key cryptography-supporting both RSA and Elliptic Curve Cryptography (ECC)-alongside hardware acceleration for cryptographic hash functions, and the facilitation of symmetric encryption methods, including AES and ChaCha20. Specifically engineered for compact, low-power devices, CryptoCell is usually integrated within systems that operate with an ARM Cortex-M or Cortex-R processor. Presently, the CryptoCell-310 variant is a key component of the nRF52840 SoC from Nordic Semiconductor, which incorporates a Cortex-M4F processor as well as the nRF9160 and nRF9161 featuring a Cortex-M33. Moreover, the advanced CryptoCell-312 has been integrated into the nRF5340 SoC, paired with a Cortex-M33 with TrustZone technology.

CryptoCell is seamlessly integrated as a memory-mapped peripheral within the system architecture. The entirety of the CryptoCell subsystem is orchestrated via a secure, memory-mapped register interface, accessible through the APB bus. This particular register plays a pivotal role in managing the functionalities

of CryptoCell, enabling the specification of cryptographic operations and the selection of cryptographic keys. For the efficient handling of input data and the retrieval of operation results, CryptoCell leverages a specialized Direct Memory Access Controller (DMAC), which operates as a master on the Advance High-performance Bus (AHB) multilayer. This sophisticated setup allows CryptoCell to perform memory operations independently of the CPU, enhancing system efficiency.

To facilitate communication with the CryptoCell subsystem, ARM provides a CryptoCell API. This API is a proprietary and confidential API owned by ARM and exclusively distributed by Nordic Semiconductor within their comprehensive nRF SDKs [2].

Hashcrypt Engine: Embedded within a wide array of NXP devices, including but not limited to the RT6XX and LPC55S6XX series, is the Hashcrypt cryptographic accelerator, leveraging the capabilities of the HASH-AES module [3]. This advanced cryptographic module significantly enhances the efficiency of symmetric cryptographic operations, such as SHA-1, SHA-2, and AES, through hardware acceleration. Control over the module is facilitated through a status and control register, which is accessible as an AHB peripheral, offering both secure and non-secure access through distinct memory-mapped registers. For efficient handling of memory operations like data reading and writing, the Hashcrypt module is equipped with a dedicated DMA. To streamline interaction with the cryptographic accelerator, NXP furnishes a proprietary Hashcrypt API within their MCUXpresso SDK, ensuring seamless integration and utilization in security-sensitive applications.

CASPER Crypto/FFT engine: CASPER [4] is a crypto co-processor to enable hardware acceleration for various asymmetric cryptographic operations such as Elliptic Curve Cryptography or RSA. The CASPER cryptographic co-processor is typically integrated with the Cortex-M33 core. Unlike other accelerators, CASPER directly connects to the main CPU via a 64-bit wide CP interface, circumventing the need for job submissions via the bus system. By leveraging the CP interface, the CASPER co-processor can be efficiently managed through the ARM architecture with MCRR instructions for initiating operations and MRRC instructions for retrieving results. This direct access allows the primary CPU to offload compute-intensive cryptographic tasks to CASPER, freeing it to either enter a low-power sleep state, operate at a reduced clock speed, or concurrently handle other system operations. Such an arrangement optimizes system resources, ensuring that the CPU can focus on parallel tasks or prepare for subsequent operations while CASPER handles the cryptographic processing.

Furthermore, CASPER's ability to access system random-access memory (RAM) through a dedicated 64-bit interface, which operates over two parallel 32-bit interleaved RAMs, exemplifies its design for efficiency and flexibility. This direct RAM access bypasses the conventional system bus, allowing for quicker data processing and reduced latency. The dual-use capability of this interface also means that when CASPER is not in operation, the RAM can be repurposed for other system needs, ensuring optimal use of resources without compromising system performance or security.

CASPER is accessed by software through a dedicated CASPER hardware interface, which is a part of the MCUXpresso SDK suite offered by NXP.

Renesas Secure Crypto Engine (SCE): The RA6M4/5 microcontrollers feature a Secure Crypto Engine (SCE) [5] as part of their design, comprising an access management circuit, encryption engine, and random number generator. This isolated subsystem ensures that internal cryptographic operations remain separated from the CPU-accessible bus, enhancing security measures. Specifically, the RA6M4/5 microcontrollers incorporate the SCE9 module, offering advanced functionality such as RSA encryption with support for up to 4096-bit keys, Elliptic Curve Cryptography on various curves, AES with up to 256-bit keys, and SHA-224 and SHA-256 hashing algorithms. On the other hand, the RA6T2 microcontrollers are equipped with the SCE5 or SCE5_B module, which provides a slightly different functionality compared to SCE9. While SCE9 offers a broader range of cryptographic operations, including RSA, ECDSA, ECC, AES, and SHA-2 algorithms, the SCE5 module supports AES encryption with 128-bit and 256-bit keys.

Both versions of the Secure Crypto Engine are isolated subsystems, ensuring their connectivity to the

internal peripheral bus through a dedicated bus interface. To further enhance security, an access management circuit is integrated between the bus interface and the accelerators. This circuit authenticates the user before granting access to the cryptographic accelerators, adding an additional layer of protection against unauthorized access. The Secure Crypto Engine operates in two distinct modes: compatibility mode and protected mode. Compatibility mode enables the use of plaintext keys and other simplifications to ensure compatibility with legacy systems. On the other hand, the protected mode restricts such simplifications, allowing only secure key injection through a dedicated serial programming interface. Notably, major software libraries like TF-M have adopted compatibility mode to maintain compatibility with existing systems.

Cryptographic Accelerator Unit (CAU): Integrated in the Kinetis K6X and K2X device family, CAU [6] provides robust support for a range of encryption algorithms, including DES, 3DES, AES, MD5, SHA-1, and SHA-256. Notably, the CAU's integration into the system differs from conventional setups. Rather than being linked to the AHB bus interconnect, it is directly connected to the Processor's Private Peripheral BUS. The Accelerator efficiently utilizes a dedicated 4 KB memory-mapped region, partitioned into two segments. The first section is writable and intended for input data, while the second section serves as intermediary storage during intricate calculations. The CAU exclusively caters to 32-bit operations and register addresses, thereby adhering to a standardized protocol. To facilitate interaction with the CAU, Freescale Semiconductor, now part of NXP, offers a dedicated software library API. This API provides developers with the necessary tools and interfaces to efficiently utilize the CAU's capabilities within their applications. A newer version of CAU, CAU3 is now available as part of the K32 L3 device series. It includes four major components: a NXP-programmable 4-stage pipeline CryptoCore, two private memories for local instruction (IMEM) and data (DMEM) storage, and a host interface register controller (HIRC) that connects the system IPS slave peripheral bus to the CAU3. From the system perspective, the CAU3 is a two-terminal device with a bus master connection to the system AHB bus fabric (for accessing crypto data in system RAM and specific slave peripheral modules) and a slave connection to the IPS bus (for the host to program and control the CAU3). To interact with CAU, NXP provides a C-function library that implements the appropriate software building blocks to execute the high-level security function as well as an optimized firmware library containing the CryptoCore code.

Cryptography Accelerator (Crypto/CryptoLite): Infineon CAT1A/C devices are equipped with PSoC 6 featuring dedicated Cryptographic Function Blocks known as Crypto [7] or CryptoLite [8] for CAT1B. These subsystems comprise hardware implementations designed to accelerate cryptographic functions and random number generation. They encompass a wide range of common cryptographic algorithms such as AES, ECC, RSA, SHA-1, and SHA-2, with CryptoLite offering a slightly reduced set of algorithms. Interaction with the subsystem is facilitated through two operation modes: the Client- Server model and Direct Crypto Core Access. It's important to note that ECP and ECDSA functionalities are exclusively available for the Direct Crypto Core Access model. In this mode, the low-level API enables direct access to the Crypto hardware. The firmware initiates and oversees Crypto operations while furnishing necessary configuration data for the desired cryptographic technique. Conversely, in the Client-Server model, the firmware initializes and manages the Crypto server, which can be executed on any core and collaborates with the Crypto hardware. This secure block conducts all cryptographic operations on behalf of the client. Access to the server is facilitated through the Inter-Process Communication (IPC) driver, with direct access being restricted. Typically, the SoC paired with this accelerator comprises an ARM Cortex-M0+ and an ARM Cortex-M4. As a result, the server instance typically operates on the Cortex-M0+, while one client instance runs on Cortex-M4, or vice versa.

While Crypto/CryptoLite provides its own API, Infineon published additional driver libraries and a hardware abstraction layer to allow integration of CAT1A/B/C MCUs with mbedTLS.

Hardware Acceleration on ESP-32: The ESP-32 offers support for SHA and AES operations, featuring two distinct modes of operation: typical and DMA working mode. In typical mode, plaintext and ciphertext are handled directly by the CPU for read and write operations. On the other hand, in DMA working

mode, the plaintext and ciphertext are managed through DMA, with interrupts being triggered upon completion of operations. Moreover, the ESP-32 is equipped with a Digital Signature Module, which enhances hardware acceleration for RSA signature generation and verification processes.

This concludes our initial investigation into the hardware accelerators currently accessible on commercial IoT platforms. Our findings highlight the vast diversity of domain-specific hardware for IoT devices and emphasize the importance of a secure software stack to ensure compatibility across a broad range of accelerators developed by various IoT manufacturers, thereby facilitating interoperability. Given that cryptographic operations are crucial for device security, and considering the computational and energy demands of these algorithms on resource-constrained IoT devices, hardware accelerators emerge as a vital solution. Based on these findings, our analysis extended into exploring APIs and standardization efforts aimed at harmonizing the multitude of vendor-specific APIs. The goal is to establish a unified cryptographic API that supports hardware accelerators, which is a critical step towards achieving interoperability across the ecosystem.

3.2 APIs for Specific HW Architectures

Integrating support for domain-specific hardware within an IoT software stack is of paramount importance, as it leads to performance optimization, enhanced energy efficiency, reduced latency, the facilitation of custom functionalities, scalability assurance, security reinforcement, and the promotion of cost-effectiveness and reliability. This seamless integration empowers IoT devices to operate at their peak potential, effectively catering to a wide array of applications.

A key facet of CROSSCON's mission lies in the provision of innovative, trusted services that contribute significantly to fortifying the security of the IoT ecosystem. Many of these trusted services directly hinge on the availability of domain-specific hardware. For instance, PUF-based authentication relies on the integration of a PUF, while hardware-assisted control flow integrity depends on the availability of control flow information and the means to enforce the integrity of the control flow through hardware mechanisms.

Conversely, certain services, such as message encryption, integrity verification, or the inference of machine learning models (e.g., for object detection), may not have a direct dependency on domain-specific hardware but can benefit significantly from their presence. This is particularly evident in the case of cryptographic accelerators and machine learning accelerators, which can greatly enhance the efficiency and capabilities of these services.

CROSSCON's objective is to establish a comprehensive specification that enables trusted services to leverage domain-specific hardware when available on a device. The ultimate aim is to delineate the required hardware components for CROSSCON's trusted services and to detail methodologies and APIs. These will facilitate the incorporation of domain-specific hardware into the CROSSCON Stack, thereby empowering trusted services to effectively utilize these resources. This serves a dual purpose. Firstly, it enables the utilization of existing hardware, such as crypto or machine learning accelerators, for trusted services. Secondly, it outlines how trusted services make effective use of new hardware components.

3.2.1 Cryptographic Accelerators

Cryptographic accelerators have become essential components of domain-specific architectures in a variety of IoT devices, enhancing the speed of vital security operations like encryption and hashing. These accelerators enable energy-efficient cryptography, even in lower-end devices. However, our analysis of existing hardware platforms reveals a diversity in design approaches among manufacturers, leading to interoperability issues. Manufacturers typically offer software tailored to their ecosystems, but appli-

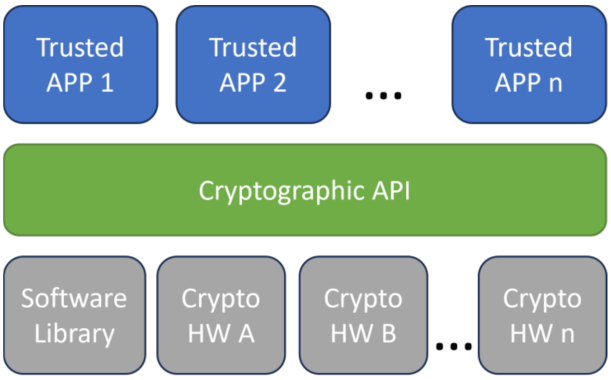


Figure 2: Unified Cryptographic Operations API

cations developed for one platform may not be compatible with the unique features of devices from other manufacturers. This often forces developers to rely on less secure, less efficient software-only solutions, bypassing the advantages of domain-specific hardware [9, 10]. To address this, developing an interoperable security stack is crucial, allowing for the integration of domain-specific hardware like cryptographic accelerators across a wide range of devices.

Figure 2 illustrates the overarching concept within CROSSCON, where we aim to implement a unified suite of APIs. These APIs will facilitate elementary cryptographic operations, including hashing, signing, and encryption/decryption of data. It is essential that this foundational cryptographic functionality be established as a baseline. This will enable the development of more complex, trusted services such as remote attestation or secure updates. Moreover, the functionality should extend beyond a mere software library. Wherever possible, it should utilize underlying hardware support for enhanced performance and security. To achieve a unified interface, our approach has been to thoroughly examine existing proposed APIs and explore various methods to integrate them within CROSSCON Stack.

Our investigation into existing APIs and their reference implementations aimed to understand the benefits and challenges of their use. The goal is to devise a solution that ensures interoperability of cryptographic operations across various devices, without necessitating the adaptation of applications, like trusted services, to the specific nuances of the underlying hardware architecture. The analysis focused on two leading cryptographic APIs: Arm's PSA Crypto API and the GP Cryptographic Operations API, with detailed findings presented subsequently.

3.2.1.1 PSA Crypto API

Arm's PSA Crypto API [11] is a component of a broader suite of APIs, developed and maintained by Arm, to facilitate a uniform programming interface for cryptographic operations across various hardware platforms. This interface is tailored to be both scalable and modular, making it ideal for devices with limited resources. It employs the concept of multi-part processing, allowing applications to process data incrementally in a streaming fashion, rather than buffering large data chunks simultaneously. Moreover, it provides optional separation of the cryptographic processors from the initiating application, as well as isolation among multiple applications. Essentially, it can implement three types of isolation: (1) No isolation, where there's no security boundary between applications and the cryptographic processor, typical for dynamically or statically linked libraries; (2) Cryptoprocessor isolation, establishing a security boundary between the application and cryptographic processor, but the processor can't communicate with other applications; and (3) Caller isolation, creating multiple application instances with security boundaries between each instance. Caller isolation involves establishing distinct application instances, each safeguarded by security boundaries to prevent unauthorized access between them. This framework enables the support of multiple independent client applications-Trusted Services, in our context-where each application interacts with shared resources like cryptographic accelerators or co-processors as if it

were the sole user. Despite multiple applications utilizing the same cryptographic hardware, caller isolation ensures that each application operates within its own secure environment. Consequently, even when applications share the same cryptographic hardware, they are prevented from accessing or interfering with each other's data or operations. This architecture enhances security by ensuring the confidentiality and integrity of application-specific cryptographic operations.

The Crypto API, defined within a single header file named psa/crypto.h, exhibits a modular design. This modularity allows for the exclusion of functions that are not necessary, such as certain cryptographic algorithms that are not utilized by any applications on the platform. Implementations of this API are responsible for providing their own version of psa/crypto.h, and they have the flexibility to implement only a subset of the full API. However, it's crucial that any API elements included in the implementation be fully accessible to an application program that incorporates this header file. The flexibility and customization offered by this modular approach make the Crypto API particularly well-suited for integration within the CROSSCON Stack. This adaptability ensures that only relevant features are included, optimizing the stack for efficiency and specific functionality.

In addition to specifying the Crypto API, Arm offers a reference implementation of the PSA API known as mbedTLS. Previously recognized as PolarSSL, mbedTLS is now maintained by Arm and is available under the Apache 2.0 license. This library positions itself as an alternative to OpenSSL, boasting a significantly smaller footprint, albeit with some limitations in functionality. mbedTLS is a comprehensive C library that implements all the specifications of the Crypto API in software, independent of any external third-party libraries. This self-contained approach enhances its usability and integration, particularly for applications requiring a lightweight yet robust cryptographic solution.

To accommodate domain-specific accelerators, such as hash accelerators or dedicated cryptographic coprocessors, PSA includes a PSA Cryptoprocessor Driver Interface specification. This facilitates a compositional build of the PSA Crypto API, allowing it to integrate seamlessly with specialized hardware.

An implementation of the PSA API consists of a core component and zero to multiple drivers. When an application makes function calls to the PSA Crypto API, these calls are processed by the core. The core then has the capability to either execute these functions directly or, if applicable, delegate specific functions to the relevant driver. This architecture enables the core to efficiently manage and route cryptographic operations, either handling them internally or utilizing external hardware acceleration as needed, thereby optimizing performance and flexibility.

The specification delineates two distinct types of drivers:

- *Transparent Drivers:* These are typically employed for hardware accelerators. For such drivers, all inputs for the functions, including keys, must be available in cleartext at the onset of each operation. This design is suited for scenarios where speed and efficiency are prioritized, and the security requirements allow for clear visibility of data at the point of processing.
- *Opaque Drivers:* Contrarily, opaque drivers are utilized for cryptographic operations within protected environments, such as secure enclaves or hardware security modules (HSMs). In these settings, sensitive information, like cryptographic keys, is confined to the protected environment. It means the keys and other secret data are exclusively accessible and usable within these secure confines, thereby enhancing security, especially in scenarios where safeguarding the confidentiality and integrity of the keys is paramount.

In the realm of trusted services, paramount importance is placed on the transparent design of drivers. This is crucial because all essential components, notably the secret keys, are administered by the trusted service.

As part of deliverable D3.4, we present an illustrative project that demonstrates the use of the PSA Crypto API. This example project highlights the encryption and decryption of data both with and without a cryptographic accelerator. It is implemented on two different development boards: one featuring

an ARM processor and the other equipped with a RISC-V processor. However, before delving into the details of this demonstrator project, we will first analyze another key candidate, the GP Cryptographic Operations API.

3.2.1.2 GlobalPlatform Cryptographic Operations API

The GP API [12] encompasses a comprehensive suite of APIs, particularly designed to enhance Trusted Execution Environments (TEEs). While its primary aim is to bolster TEEs, it notably includes a specialized Cryptographic Operations API, which is an integral part of the TEE Internal Core API. OP-TEE, maintained by Trusted Firmware, stands as the standard implementation of the entire suite, including the Cryptographic Operations API.

This framework establishes specific system calls for all cryptographic activities. These calls not only handle the conventional tasks associated with the user/kernel interface-such as parameter verification and memory buffer transfers between user and kernel spaces-but also activate a proprietary layer: the Crypto API, outlined in the crypto.h header. This API fulfills two crucial roles: it allows for the deactivation of certain algorithms at compile-time to optimize space, and it supports the provision of alternative implementations, such as hardware-accelerated versions.

Within the TEE Core, there is a single, active default implementation of the Crypto API, primarily derived from LibTomCrypt, a versatile, open-source cryptographic toolkit. To facilitate the incorporation of cryptographic hardware accelerators or co-processors, the framework provides a Generic Cryptographic Driver interface. This interface bridges the TEE Crypto APIs with the hardware interface, enabling the development of custom drivers to link OP-TEE with available hardware accelerators. Nevertheless, to utilize these hardware-accelerated versions, the standard software-based implementation must be disabled in OP-TEE's configuration settings.

3.2.2 Physically Unclonable Function

Physically Unclonable Functions (PUFs), first described in 2002 [13], are a security technology that provides a way to secure devices by leveraging the inherent and unique physical variations that occur during the manufacturing process. These variations are unpredictable and effectively impossible to replicate, making each PUF-enabled device intrinsically unique. At its core, a PUF exploits the minor, random physical differences found in electronic components to generate a unique identifier or response to a given challenge (a specific input or question). This response can then be used for a variety of security applications, including authentication, secure key generation, and anti-counterfeiting measures.

The principle behind PUFs is somewhat analogous to human fingerprints. Just as no two fingerprints are exactly alike, no two PUFs, even those manufactured in the same batch, will produce the same response to a given challenge. This uniqueness is due to microscopic differences in the material properties and manufacturing anomalies that occur naturally and unpredictably. This Challenge and response pairs (CRPs) in PUF technology are generated through a process where a specific input (the challenge) is presented to the PUF, and the device produces an output (the response) based on its unique physical characteristics. These pairs are critical for verifying the authenticity and integrity of the device, as only the genuine device with the specific PUF can produce the correct response to a given challenge. This mechanism is utilized in various security protocols to ensure that only authorized devices can access certain features or data, leveraging the unclonable nature of PUFs to prevent tampering and duplication by malicious actors.

PUFs are typically divided into two categories, known as strong and weak PUFs, based on their capacity to generate Challenge-Response Pairs. Weak PUFs have a limited ability, producing only a single or at best a linear number of CRPs. This limitation makes them suitable for applications where a small, fixed set of responses is sufficient, such as device identification or key storage. On the other hand, strong PUFs are capable of generating a vast number of responses, exponential in relation to the length of the

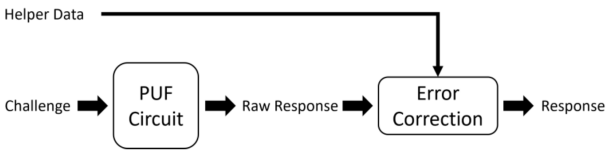


Figure 3: High-Level PUF Architecture

challenge. This extensive range of responses allows strong PUFs to be utilized in more complex security applications, including dynamic authentication processes where a large and varied set of CRPs enhances security by making it difficult for attackers to predict or replicate responses. The distinction between strong and weak PUFs thus lies in their respective capabilities to produce CRPs, a key factor determining their application in securing digital systems.

Figure 3 illustrates a generic PUF architecture where the PUF circuit is fed a challenge, subsequently generating a raw response. Given the inherently noisy nature of PUFs, various architectures incorporate error correction or fuzzy extraction techniques to ensure the production of stable responses. These error correction mechanisms necessitate supplementary information, typically generated during the PUF's enrolment phase at the time of manufacturing. This information is then combined with the raw PUF response within the error correction algorithm to produce a stable, reliable response.

Within the CROSSCON Stack, our objective is to introduce an innovative authentication service powered by PUF, alongside a two-factor authentication system that employs PUF technology as an additional layer of security. Our approach is designed to be PUF-agnostic, accommodating both weak and strong PUFs to ensure broad compatibility and flexibility. However, our primary emphasis is on leveraging strong PUFs due to their superior security features and versatile application potential. PUFs are already employed in some devices; however, as of now, there is no unified API available, only proprietary solutions, which are discussed below.

NXP PUF module: As part of the MCUXpresso SDK, NXP offers a comprehensive peripheral driver for the PUF across all its devices. This includes an API for enrolling the PUF, generating helper data (referred to as the activation code), producing PUF responses, and key storage using PUF hardware. Additionally, the SDK provides a method to block re-enrollment of the PUF, an essential feature considering NXP's exclusive use of SRAM PUF, a weak PUF that offers a limited set of potential challenge-response pairs and is susceptible to exhaustive querying if not adequately safeguarded.

Xilinx Embedded Software Development (embeddedsw): Within embeddedsw [14], Xilinx provides an API for the PUF of all Zynq UltraScale+ devices. This API provides functionality to enroll the PUF and generate the necessary helper data, generate a PUF response, and regenerate already enrolled keys and set the necessary eFuses to prevent re-enrollment of the PUF. It's important to note that the design of this API doesn't entail direct access to the PUF for tasks such as PUF-based authentication. Instead, its primary function is to enable the encryption and decryption of user-provided data using the PUF-generated device key.

While PUFs are used by further vendors, such as within Maxim's Integrated ChipDNA [15], Pufsecurity's PUFrt [16] and PUFcc [17], or Synopsys Software-based PUF IP [18], these represent proprietary PUF solutions catering to diverse use cases. Notably, the vendors do not disclose an openly accessible PUF API.

3.2.2.1 Trusted Service Interface

For the authentication schemes, we thus define the implementation-agnostic interface as follows:

> "A function that, given an input challenge, returns a reproducible fixed-length array of characters representing the PUF's physical response (after any necessary error correction).

> For weak PUFs, the challenge does not affect the physical behavior of the device, but is incorporated into a virtual response via a secure hash function."

It is well established in the literature that the inclusion of hashing mechanisms can pose computational or energy concerns on highly resource-constrained platforms such as ultra-low-power microcontrollers or passive RFID tags [19] [20]. This naturally limits the set of devices where such authentication schemes can be deployed. In practice, system designers often face a tradeoff between using a physically stronger PUF (e.g., one with many CRPs) or relying on a more computationally capable platform to support cryptographic post-processing. However, our focus here is on standardization rather than absolute minimalism. Since we target devices that support a Trusted Execution Environment (TEE), such as those compliant with the GP specification, we assume the presence of cryptographic capabilities.

The PUF's physical response shouldn't be exposed to system memory however it might be impossible in some implementations. In this scenario then the PUF's physical response should be flushed immediately after hashing with challenge.

The strength of this definition is critically dependent on the entropy of the PUF response. Specifically, because the PUF response is concatenated with the public challenge and then hashed, an attacker with knowledge of the challenge and hash could mount a brute-force or preimage attack on the PUF response.

To resist such attacks, the PUF response should have a minimum entropy of λ bits, where λ is the security parameter (typically $\lambda = 128$ or higher for modern cryptographic security).

Let:

- r be the PUF response (after error correction)
- C be the public challenge
- $h = \text{SHA256}(r||C)$

Then, to make brute-force inversion of r infeasible:

$$2^{H(r)} \gg 2^{\lambda} \quad \Rightarrow \quad H(r) \geq \lambda$$

Assuming the PUF outputs uniformly random bits:

$$\text{Minimum recommended length of } r \geq \lambda \text{ bits}$$

Thus, for $\lambda = 128$, the PUF response must be at least 128 bits (16 bytes) long. Shorter responses reduce the effective security, allowing attackers to attempt preimage attacks by enumerating all possible PUF outputs.

Non-Uniform PUFs: In practice, many PUFs exhibit non-uniform or biased behavior, where some bits are more likely to be 0 or 1, or where output bits are correlated. In such cases, the actual entropy of the response is lower than its bit length.

Let n be the number of output bits and let each bit r_i have a bias $p_i = \Pr[r_i = 1]$. The total entropy can be approximated using the sum of binary entropies:

$$H(r) = \sum_{i=1}^{n} H_{\text{bin}}(p_i) \quad \text{where} \quad H_{\text{bin}}(p) = -p \log_2 p - (1-p) \log_2 (1-p)$$

If the PUF exhibits uniform behavior ($p_i = 0.5$), then $H_{\text{bin}}(p_i) = 1$ for all i, recovering the earlier result $H(r) = n$.

For biased PUFs ($p_i \neq 0.5$), the entropy per bit decreases, and a longer response is required to achieve the target entropy:

$$n \geq \frac{\lambda}{\bar{H}_{\text{bin}}} \quad \text{where} \quad \bar{H}_{\text{bin}} = \frac{1}{n}\sum_{i=1}^{n} H_{\text{bin}}(p_i)$$

For example, if each bit has a bias of $p_i = 0.8$, then $H_{\text{bin}}(0.8) \approx 0.72$, and the response length must be at least:

$$n \geq \frac{128}{0.72} \approx 178 \text{ bits}$$

This highlights the need to evaluate the statistical properties of the PUF and adjust the output length accordingly to maintain cryptographic security.

3.2.2.2 GlobalPlatform Perspective

GP Core API does not define a dedicated API for accessing physically unclonable Functions (PUFs) [21]. However, the GP specification does provide mechanisms for exposing hardware peripherals to Trusted Applications (TAs) via the TEE, particularly under the notion of *TEE-ownable devices*. According to GP's TEE Internal Core API specification:

> "Peripherals that are temporarily or permanently isolated from non-TEE entities, managed by the TEE, and fully usable by a TA through the APIs the TEE offers. These devices are described as TEE ownable."

As of writing this chapter, this principle is sufficient to enable PUF integration in systems that can isolate the PUF peripheral to secure world execution. For instance the PUF module can be made accessible exclusively within the secure world and subsequently wrapped in a Trusted Service for use by Trusted Applications (TAs).

The diversity of PUF implementations, including analog, optical [22], and other non-digitally addressable variants, makes it unlikely that a universal, low-level Core API for raw PUF access would be appropriate. Instead, standardization efforts should consider defining higher-level abstractions that account for implementation variability while supporting common use cases such as authentication and device-unique key derivation. These abstractions would need to address not only hardware access but also statistical characteristics (e.g., entropy bounds, uniformity, bias correction), as well as optional enrollment and fuzzy extraction mechanisms.

Crucially, in most threat models it is undesirable for a Trusted Application (TA) to have access to the raw PUF response, since this would allow TAs to misuse or exfiltrate the physically unique secret. A more secure and practical model, especially for general purpose TEEs, would be to to allow each TA to obtain a reproducible, volatile secret derived from the PUF, scoped to its identity. This could be achieved by hashing the PUF response together with the TA's UUID or another platform-assigned namespace.

We propose that future versions of the GP specification introduce a standard interface within the TEE Peripheral API for such functionality.

One proposal is to extend the `TEE_PERIPHERAL_TYPE` enumeration (e.g., Table 9-3 in GP Core v1.3.1) to explicitly include PUFs, enabling BSP providers to expose PUF services. Thus already existing mechanisms in GP Core like `TEE_Peripheral_Read` might be then used to get a bounded-length, high-entropy secret unique to the calling TA. This mechanism could be implemented either in hardware (e.g., fused into the SoC as part of the PUF peripheral logic) or as a trusted service provided by BSP. The latter offers greater flexibility and supports a wider range of PUF technologies, provided the TEE ensures isolation and enforces access policies.

3.2.2.3 Prototype

In the prototype made for UC1.1, we encapsulate the PUF interaction within a Trusted Application, adhering to GP's IPC and lifecycle management model, while remaining agnostic to the specific PUF architecture or post-processing logic.

Due to the lack of readily available hardware with native PUF support, the prototype targets the NXP LPC55S69 board, which exposes an SRAM-based PUF interface. However, the platform lacks support for any recognized TEE implementation. As a result, the Trusted Application is implemented as a Zephyr "VM" running on top of the CROSSCON Hypervisor.

Although the `TEE_Peripheral` API is defined in the GP specification, it is worth noting that it remains unimplemented in open-source TEEs such as OP-TEE and MTower. As such, our prototype does not rely on this interface, but instead simulates its intended semantics through the following mechanisms:

- **Isolation of the PUF module:** Access to the PUF hardware is restricted by the CROSSCON Hypervisor's static partitioning, which grants exclusive access to the Trusted Application running in the Zephyr VM. This ensures that non-secure components and other VMs cannot observe or interfere with the PUF interface.
- **Identity-scoped secret derivation:** Although in our prototype the TEE is tightly coupled with a single Trusted Application, effectively forming a single security domain, we still apply identity-scoping as a proof of concept. The challenge–response generation is performed using a two-stage derivation process: first, the PUF response is hashed together with the TA's UUID to bind it to the application identity. Then, the result is hashed again with the challenge. This produces a volatile, reproducible secret that would support per-TA isolation in a true multi-tenant TEE.
- **Abstraction over PUF technology and post-processing:** The Trusted Application does not depend on any particular PUF implementation or post-processing pipeline. It supports a variety of PUF types—including noisy or low-entropy variants—so long as the response is stable enough for keyed derivation. Post-processing techniques such as bias correction, helper data, or fuzzy extraction may be used internally to stabilize the raw PUF response. This abstraction simplifies portability across platforms and PUF technologies.
- **Partial GP compliance:** To improve portability, the commitment and proof generation logic is implemented using the GP Core Crypto API abstraction. This structure enables adaptation to other TEEs, platforms, or cryptographic libraries with relatively low integration effort. While Cortex-M development constraints require some additional platform-specific work, we consider the resulting architecture to offer "relative ease" of adaptation. More information on client-side communication is documented in the D3.3 document.

Below is a portion of our implemented interface. Porting to a different PUF would involve providing a C source file that implements these functions for the target hardware. This could be further wrapped within `TEE_Peripheral` calls.

```
#ifndef PUF_HANDLER_H
#define PUF_HANDLER_H

#include "tee_internal_api.h"

#ifndef CHALLENGE_SIZE
#define CHALLENGE_SIZE 32
#endif

#ifndef PUF_RESPONSE_SIZE
```

```
#define PUF_RESPONSE_SIZE 128
#endif

/**
 * @brief Initializes the PUF module.
 *
 * May perform enrollment or hardware setup depending on the implementation.
 *
 * @return TEE_SUCCESS on success, or error code on failure.
 */
TEE_Result init_puf(void);

/**
 * @brief Retrieves a derived key from the PUF, optionally scoped by a label.
 *
 * For weak PUFs, this label may be used as a salt in a hash function
    together with
 * the intrinsic PUF response (e.g., TA UUID or purpose).
 *
 * For strong PUFs, the label may be used internally by the hardware to
    derive
 * a distinct key using built-in diversification mechanisms.
 *
 * @param label Optional label or purpose string (can be NULL for default
    key).
 * @param label_len Length of the label in bytes.
 * @param puf_key Output buffer for the derived key (must be
    PUF_RESPONSE_SIZE bytes).
 * @return TEE_SUCCESS on success, or error code on failure.
 */
TEE_Result puf_get_key(const uint8_t *label, size_t label_len, uint8_t
   *puf_key);

/**
 * @brief Securely wipes the key from memory.
 *
 * Overwrites the buffer with zeros and verifies wipe.
 */
TEE_Result puf_flush_key(uint8_t *puf_key);

_Static_assert(PUF_RESPONSE_SIZE >= 128, "PUF Response size should be
   greater than 128 bits");

#endif /* PUF_HANDLER_H */
```

3.2.3 Hardware Primitives for Environmental Fingerprinting

Similar to physically unclonable Functions, which exploit hardware imperfections occurring during the manufacturing process of integrated circuits to generate unique responses (digital fingerprints) to a given challenge, wireless network transmitting hardware is also influenced by minor fluctuations taking place during the manufacturing process. These fluctuations may include variations in power amplifiers, mixer

imbalances, or oscillator imperfections.

Within the CROSSCON Stack, we aim to utilize Wi-Fi enabled devices to fingerprint Wi-Fi transmitters within a predefined location, including its layout, and quantity, and type of transmitters to generate a digital environmental fingerprint of these environmental transmissions. The fingerprint embeds unique hardware imperfections of the transmitters, including interference with the location layout, which may affect signal quality due to events such as reflection or deflection, thus resulting in a unique fingerprint of the environment. However, unlike physically unclonable Functions, there is no specific response requiring a corresponding challenge but instead the imperfections manifest as observable patterns in the transmitted signals.

Therefore, this environmental fingerprint can serve as an additional layer of authentication by enabling a device to verify its location, particularly for security-sensitive operations such as firmware updates. Due to the uniqueness of the fingerprint, a remote attacker would ultimately need to resort to brute force or guessing.

As Wi-Fi is a multi-party communication protocol, the communication is split into multiple channels, which, in turn, are further broken down into sub-channels of equal bandwidth [23]. Each subchannel occupies a small portion of the radio frequency spectrum with its own frequency, with each frequency being impacted by the device's environment in a different way. These impacts can be measured as the changes in amplitude and phase shift over time.

The latest Wi-Fi standards allow multiple antennas to be used for transmit and receive purposes. Multiple antennas are used together to steer the radio beam in the direction of the connected device, increasing throughput. Currently, multi-antenna configurations are not typically found on Internet of Things (IoT) devices.

Many Wi-Fi devices regularly measure the signal's amplitude and phase properties for received packets in order to analyze their physical environment and to assess the Channel State. This information is used to optimize the transmission process. The collected data is known as Channel State Information (CSI) and is the perfect candidate for creating a fingerprint of a device's environment. Although many devices collect CSI data, only a few chips can forward this information to the user, often requiring modified firmware to do so.

Consequently, a device must be equipped with Wi-Fi hardware which allows the collection of CSI data to create a fingerprint of its environment. Further, a dedicated memory section must be allocated to the trusted service, which is utilized to store and write the collected information. This area is accessible only by the relevant trusted service and is protected against unauthorized access. Eventually, the collected data must be transmitted to an external party via an end-to-end secured channel to be processed into a fingerprint.

3.2.3.1 Interfaces for Trusted Services

From the perspective of a trusted service that uses CSI data for environment fingerprinting, we define the implementation-agnostic interface as follows:

> "A function that, given a Wi-Fi channel, the channel bandwidth, a collection timeout, and optionally a Media Access Control (MAC) address filter and a number of samples to be collected per device, collects CSI samples and returns the number of collected samples as well as the samples themselves."

A peripheral which measures CSI data should provide more than just the amplitude and phase shift values in a sample. The packet's origin MAC address, an identifier for the spatial stream the packet was transmitted on, and a timestamp for when the packet was recorded must be included with the CSI data.

- **Origin MAC Address:** The CSI values are recorded for each packet, and a packet originates from a device connected to the network. As such, it is identified by the source MAC address provided in the packet. Providing the MAC address alongside the measured CSI values enables mapping the values to their origin device, which leads to more fine-grained insights into the physical environment.
- **Spatial Stream Identifier:** A device equipped with multiple antennas exchanges each packet on exactly one antenna via exactly one spatial stream. Different spatial streams have different physical properties because the antennas are located in different positions. As a result, it is important to know which spatial stream was used for the transmission of a single packet to map the measured properties to the spatial stream.
- **Timestamp:** Some devices send packets periodically, while others show bursty behavior. This information can be valuable for deciding how the collected CSI measurements are used. As such, each measurement must contain the timestamp at which it was taken. This timestamp could be either absolute or relative to the start of the recording.

The size of a single CSI sample is dependent on the Wi-Fi network configuration. Wi-Fi offers different options for bandwidth to choose from during configuration, and the bandwidth decides the number of sub-channels used for data transmission. For example: A 20 MHz channel is divided into 64 sub-channels, whereas 80 MHz result in 256 sub-channels. Amplitude and phase shift are measured for each of those sub-channels. As each packet is received on only one spatial stream, the number of antennas found in a multi-antenna systems has no influence on the sample size.

The number of CSI samples generated in a fixed time interval is mainly influenced by the Wi-Fi environment. Because CSI values are measured for every packet received, a more busy wireless network will result in more samples collected. This can be caused either by more devices connected to the network, which periodically send similar amounts of data, or by devices on the network sending data more frequently. Moreover, some devices might send packets in bursts, while others transmit packets periodically. Thus it is paramount to include the packet's origin MAC address and the measurement timestamp within the collected sample in order to gain more insight into the environment.

3.2.3.2 GlobalPlatform Perspective

Similar to PUF, the GP Core API does not provide a direct interface for retrieving CSI data from Wi-Fi hardware [21]. Unlike PUF, Wi-Fi chipsets are generally not only used by Trusted Execution Environments (TEEs), but also by Rich Execution Environments (REEs) for accessing a wireless network. Currently, there is no device available which can fulfill both tasks simultaneously, so while general access to the device is shared between the environments, only one can access it at a given point in time. This makes it difficult to categorize according to the GP Core API specification for peripherals: If it is controlled by the REE, the REE could deny access by the TEE, leading to security-critical tasks relying on the CSI data not being executed. If controlled by the TEE, a sudden network unavailability might negatively impact the application payload running on the REE.

Ideally, the collection of CSI samples and Wi-Fi network connectivity could occur at the same time and with minimal impact on each other. As such, instead of thinking of the chipset as one peripheral, it could concievably be split into two virtual peripherals: One peripheral is controlled by the REE and provides Wi-Fi connectivity. The other peripheral is TEE-controlled, provides CSI sample collection, and can be considered `TEE-ownable` according to the GP Core API specification. This also enables separate CSI measurement peripherals which only provide CSI collection and no network connectivity. It also serves as an additional measure for privacy: As alluded to above, CSI data can be used for device localization, and, as a consequence, it should be protected as much as possible. Limiting access to it from within a TEE prevents a compromised REE from gathering sensitive information about the device's environment.

Because the structure of CSI samples is a direct consequence of the official Wi-Fi standard, it is possible to create a unified interface for how CSI samples are received from peripherals. This interface should facilitate the configuration of recording parameters (e.g. Wi-Fi channel, channel bandwidth) in one direction and streaming samples in the other direction. Since the GP Core API already provides functions for interacting with peripherals, specifically writing a peripheral's state and reading data from it, we propose extending the list `TEE_PERIPHERAL_TYPE` with a CSI source peripheral. The function `TEE_Peripheral_SetState` could be used for configuring the collection parameters, and `TEE_Peripheral_Read` provides a way for streaming the collected CSI samples back. It should be noted that this necessitates a buffering mechanism for collected, but not yet read samples. This buffer could be implemented either in hardware or in software.

3.2.3.3 Prototype

The prototype built for UC 1.2 provides a Trusted Application (TA) which triggers the collection process of CSI data through mechanisms compliant with the GP Core API. At the same time, the CSI collection mechanism can be adapted to work with a multitude of different Wi-Fi chipsets and drivers.

The platform chosen for this prototype is the Raspberry Pi 4. Its processing cores are based on the ARM Cortex-A design, and the OP-TEE TEE is available for this board, allowing easy development of GP API-compliant TAs. Furthermore, a modified firmware is available for its Wi-Fi chipset which allows the extraction of CSI data as part of the Nexmon project [24][25]. While accessing the wireless network is not possible during CSI sample collection, this is acceptable in this use case as an active Wi-Fi connection is not required.

However, the Nexmon firmware can only be loaded from a Linux environment, and communication with it requires special kernel modules. To accommodate for this, we host a separate Linux Virtual Machine (VM) on the CROSSCON Hypervisor (Hypv.) with the sole task of mediating the communication between OP-TEE and the Nexmon firmware. Because of that, its hardware access is restricted to a Hypv. IPC between it an OP-TEE and to the Wi-Fi chipset, besides the CPU and memory to sustain a Linux environment, and can generally be considered secure.

OP-TEE does not implement the `TEE_Peripheral` API as described by GP API. To simulate the intended behavior, a pseudo Trusted Application (pTA) is included in OP-TEE Operating Systems (OS) to facilitate communication with the Linux TA. It can read and write the IPC's shared memory in a controlled fashion and serves as an additional layer between a TA and the shared memory. Furthermore, since the mechanism of one TA calling another is part of the GP Core API implemented by OP-TEE, this communication can generally be considered to be compliant with the GP Core API. Portability between hardware platforms is achieved by re-implementing the pTA to use platform-specific methods for CSI sample collection, possibly circumventing the separate VM altogether.

The commands provided by the pTA are wrapped with C functions. An extract of the corresponding header file is given below.

```
TEE_Result check_if_response_available(
    uint8_t* available,
    uint8_t* return_reason,
    uint32_t* num_samples_collected
);

TEE_Result read_samples(
    uint8_t* buffer,
    uint32_t num_bytes_to_read,
    uint32_t read_offset_bytes,
    uint32_t* bytes_read
```

```
);

TEE_Result set_mac_filter(
    uint8_t* mac_addrs,
    uint8_t num_macs
);

TEE_Result disable_mac_filter(
    void
);

TEE_Result set_recording_parameters_and_start(
    uint8_t wifi_channel,
    uint8_t wifi_channel_bandwidth,
    uint16_t recording_timeout,
    uint8_t num_samples_per_device
);
```

The returned samples contain not only the CSI data, but also the origin MAC address, the signal strength with which the packet was received (RSSI), the seconds passed between start of the recording & sample received, and the Wi-Fi spatial stream. This information can be used as additional input into the fingerprinting process.

4 Hardware Security Mechanisms and Guarantees

A common trend in the industry is to perform multiple tasks of different criticality on the same system-on-chip (SoC). This allows companies to reduce costs and simplify development, but at the same time exposes a larger attach surface as more of the system resources are shared. The isolation of shared resources is a hard problem, especially as any information that the software can obtain can potentially be used to learn something about other parts of the system, as was demonstrated many times before, for example, through leveraging side-channel attacks [26] [27] [28]. This information can often be obtained from lower levels of the stack, which highlights the need to consider the entire hardware–software stack when aiming to provide meaningful software isolation.

As part of the effort to make software isolation easier and to reduce the attack surface, we have developed a RISC-V based solution that helps us to isolate software through the entire hardware-software stack while allowing us to utilize hardware as much as possible, through sharing hardware modules (e.g. hardware accelerators). Furthermore, we have developed a run-time exploit mitigation mechanism that leverages the ARM Memory Tagging Extensions (MTE) to provide a persistent and deterministic defense against stack-based spatial memory errors, while preserving the integrity of pointer tags, called TAGShield.

We start this chapter by describing a CROSSCON SoC (Section 4.1) that leverages our solution for cross hardware-software stack isolation and continue with describing TAGShield (Section 4.2).

4.1 CROSSCON SoC

CROSSCON SoC is a SoC design that provides RISC-V execution environment for mixed criticality IoT devices that require strong security guarantees, flexibility, small code size and low power consumption. CROSSCON SoC is based around Beyond Semiconductor's BA51 (RISC-V) core with HW extensions that enable isolation of software by running it inside of hardware-enforced software-defined virtualization-based TEEs. Furthermore, CROSSCON SoC allows sharing of HW modules connected to SoC interconnect between TEEs without compromising isolation by using PG: a hardware module that, when placed between SoC interconnect and the HW module, allows better access control to the HW module.

In this chapter, we describe the CROSSCON SoC and the solutions that we leverage to obtain cross hardware-software stack isolation.

4.1.1 Hardware-software stack isolation

Any information that the software can obtain from the system it is running on can potently be used to learn something about the other parts of the system that the software should not have access to. Such information can often be obtained from lower levels of the system, as this was already demonstrated many times before [26] [27] [28]. This shows that, in order to obtain any meaningful isolation, we need to consider the entire hardware-software stack when enforcing isolation.

One way we can approach this problem is by tracking how the information flows through the layers of the hardware-software stack and, for each layer, ensure that no information can be leaked. Figure 4 shows a high-level architecture of a generic SoC with a single RISC-V processing core, highlighting a basic separation of HW components between different levels according to traditional view of the SoC architecture. In the first level, we want to isolate software while sharing several cores and memory; on the second level, we want to isolate transfers while sharing interconnect, and, on the third level, we want to isolate users of individual HW modules while allowing the modules to be shared.

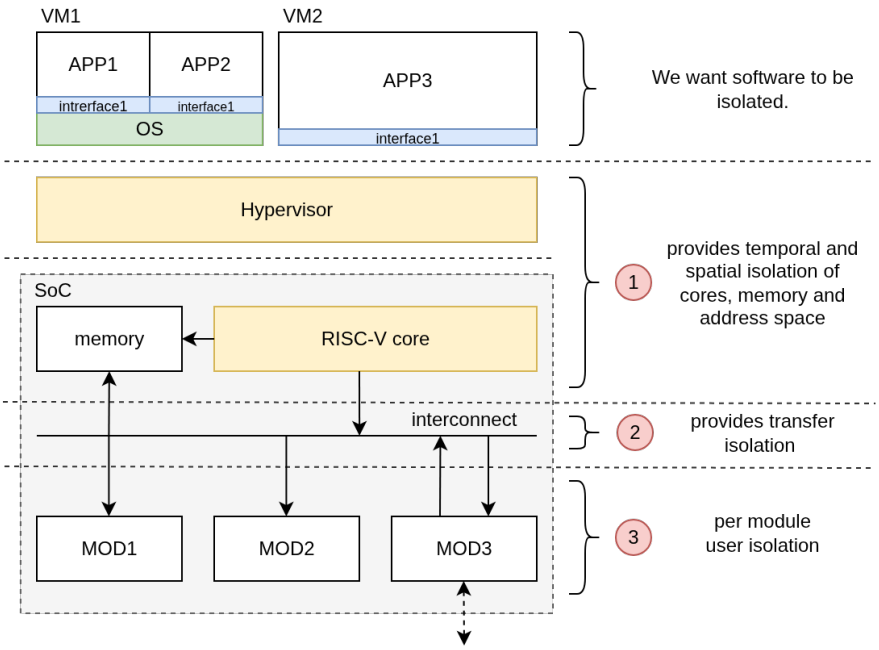


Figure 4: High-level overview of a typical security-focused SoC and guarantees that each level should provide

From this perspective, the problem of isolating software boils down to: (1) isolating information on each level of the stack, while (2) making sure that interaction between the levels does not break the isolation.

Performance and utilization: Note that any isolation mechanism that we use should have a minimal impact on performance and should allow full utilization of the hardware. Otherwise, the mechanism will not be adopted in practice, as it would make the final solution more expensive, which contradicts the initial motivation for why companies are leaning towards using a single SoC for mixed-criticality software.

Caller isolation: An example of industrial application of our solution that provides cross hardware-software stack isolation is enforcement of **caller isolation** in the context of the ARM's PSA Crypto API [11]: an interface that provides key management and cryptographic primitives. By ensuring caller isolation of PSA Cryto API library we ensure that, for example, keys cannot be exfiltrated and that the planitex stays private even if multiple users are using the library and the underlying HW resources, e.g. cryptographic accelerator.

4.1.2 Architecture

We have designed the CROSSCON SoC to allow the information to be isolated on each individual level while tracking who "owns" the information by using a Domain ID (DID) that is propagated through all the levels of the stack.

Figure 5 shows the basic architecture of the CROSSCON SoC that consists of the BA51-H core, main memory (QMEM), AHB interconnect protected by Perimeter guard, and several HW modules connected to the interconnect, including AES-GCM accelerator, smaller SRAM, and two UARTs. According to Figure 4 we can divide the CROSSCON SoC into three layers. In the first layer, we have a CROSSCON Hypervisor and BA51-H core together with QMEM that allows isolation of software into hardware-enforced software-defined TEE. In the second layer, we have an AHB interconnect protected by PG that allows us to restrict

the communication on the interconnect only to hardware modules that belong to the same domain. In the third level, there are hardware modules connected to the interconnect that together with PG can be shared between different domains while preserving isolation; for example, the AES-GCM accelerator that allows context switching on block granularity. In the following chapters, we describe the main components of the SoC and how they interact with each other.

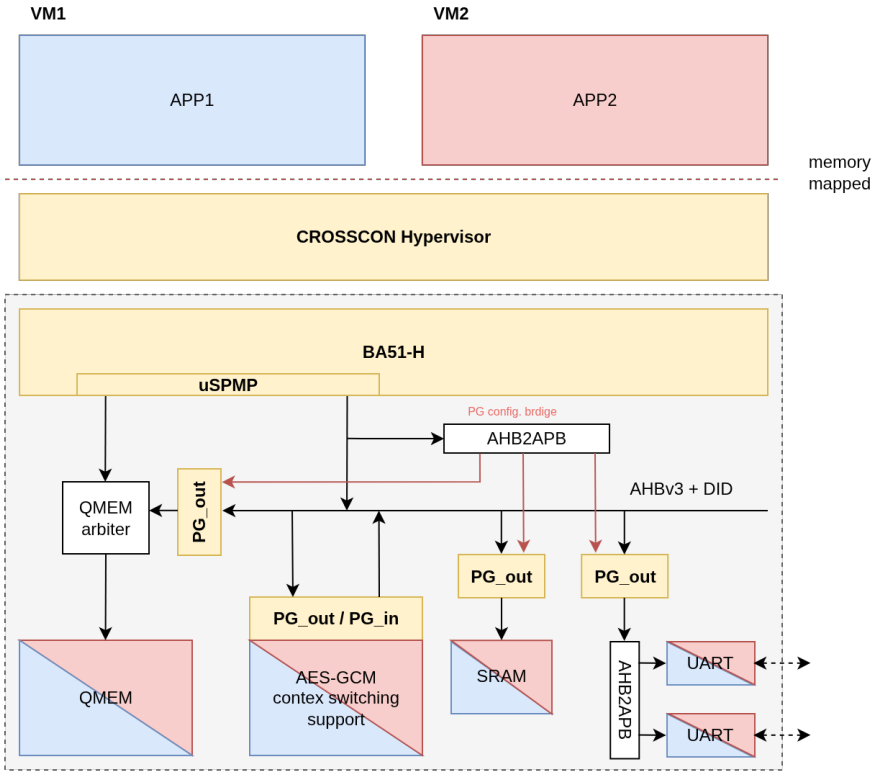


Figure 5: High-level architecture of the CROSSCON SoC

Note that the CROSSCON SoC can be used as the core architecture around which we can construct a secure SoC by adding additional processing cores and hardware modules. The current selection of modules mainly serves as a representative setup where we can demonstrate how isolation is enforced across levels while providing some of the essential features of a secure and efficient SoC.

4.1.3 Threat model

A detailed description of a threat model that we have considered is described in Chapter 7.2.3 (The threat model) of D2.3 [29]. But briefly, we assume that the attacker has control of one of the TEE's, also called virtual machines (VM), and that he tries to learn something about other TEEs running on the system, e.g. either by finding the vulnerability in the isolation provided by BA51-H (uSPMP, interrupts, etc.) or try to get some information through shared hardware modules, for example, the AES-GCM accelerator.

4.1.4 BA51-H

The BA5x is a line of RISC-V cores provided by Beyond Semiconductor [30]. As part of the CROSSCON project, we have extended BA51 [31] in a novel way to obtain simple and efficient RISC-V core level isolation.

BA51 core: The BA51 (Figure 6a) is a configurable, low-power, deeply embedded RISC-V processor IP core that implements a single-issue, in-order, 2-stage execution pipeline and supports the RISC-V 32-bit

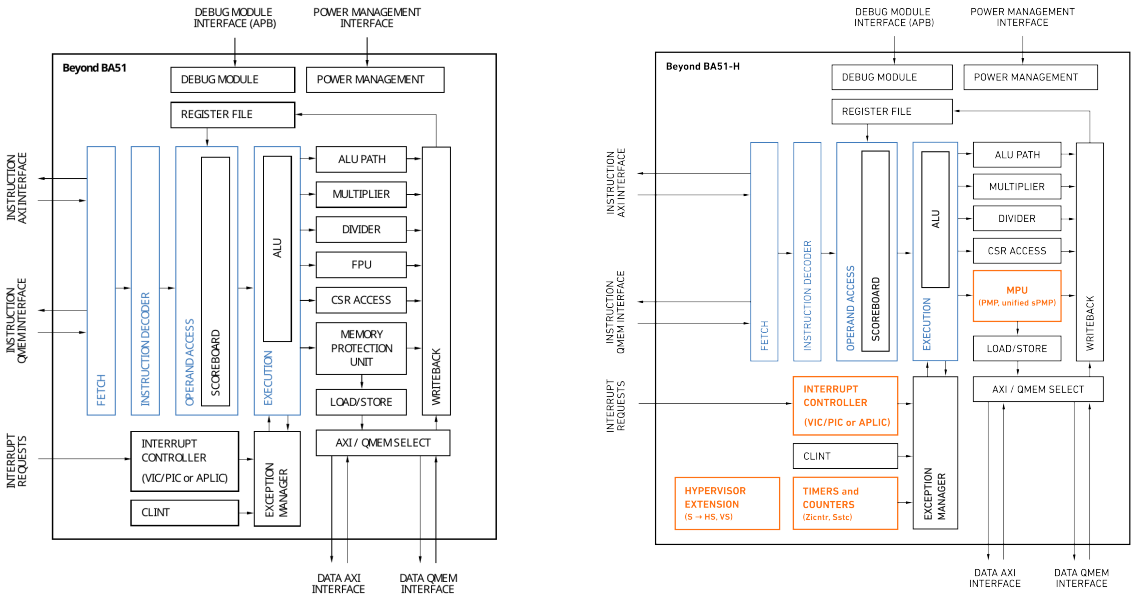
base integer instruction set (RV32I) or the 32-bit base embedded instructions set (RV32E). The processor core can be configured to meet different application requirements. For instance, it can optionally support user and supervisor modes (U-mode and S-mode respectively), as well as the ISA extensions for Compressed Instructions (C), Integer Multiplication and Division Instructions (M), Atomic Instructions (A), User-Level Interrupts (N), Control and Status Register (Zicsr), and Instruction-Fence (Zifencei); where support for the single-precision floating-point (F) ISA can also be added. Furthermore, the BA51 supports software and timer interrupts and up to 64 external interrupt lines; where the elapsed time from when an external interrupt is asserted until the first instruction in the resolved interrupt handler can be issued is just 4 clock cycles. The BA51 is designed for low energy consumption. It is compact and enables advanced power management. Under its minimal configuration, the processor size is just 16k gates, and even when most of the optional features are enabled, it is about 50k gates. The BA51 can effectively replace existing 8-bit and 16-bit microprocessor units (MCUs) or be used as a secondary, housekeeping, or peripheral controller processor in complex SoC designs. It is suitable for a wide range of deeply embedded applications such as mixed-signal embedded processing (e.g., SERDEScontrol), wireless communications ICs (e.g., Bluetooth, Zigbee, or GPS), and industrial MCUs.

BA51-H: As part of the CROSSCON project, we have extended the BA51 core (Figure 6) with additional extensions that allow efficient isolation of software running on the core. We have added the following extensions: unified (two-stage) S-mode Physical Memory Protection (SPMP), Hypervisor extension without virtual memory support, S-mode timers (Sstc) and Advanced Platform-Level Interrupt Controller (APLIC). The extensions allow a hypervisor to establish hardware-enforced, software-defined virtualization-based TEEs: a novel TEE design that leverages hardware virtualization primitives to isolate software (e.g. RTOS and applications) into dedicated virtual execution environments (VMs). A solution similar to ARM's TrustZone but more general because it allows one to establish several orthogonal TEE instead of relying only on two worlds. To the best of our knowledge, the BA51-H is the smallest silicon area RISC-V core with hardware virtualization support featuring unified SPMP, Sstc, and APLIC extensions. The silicon area footprint of BA51-H is mainly driven by two factors: the processor core logic size (gate count) and the size of the memory, which were the main factors when extending the BA51 core. We balanced our design in terms of additional gates required for the new extensions, the required memory size (i.e. code size reduction of the hypervisor) and the additional time overhead of the hypervisor required for the virtualization. This required moving some functionality from software to hardware (i.e. interrupts delegation) and vice versa.

Hardware resources: Table 1 shows the synthesis results of BA51 compact configuration (CC) and BA51-H feature-rich configuration (FRC). We have synthesized both designs using the ASIC synthesis tools on an advanced node process with a 100 MHz nominal timing constraint. As is usual and because we are interested in the relative size of a particular feature, we use gate count as a process-independent indication of area. Furthermore, we estimate that one bit of SRAM has the size equivalent to one gate as a process-independent measurement of SRAM area. The "BA51 compact" is a compact setup of BA51 with PIC, M-mode only, and C, E, Zicsr extensions, where "BA51-H FRC" is a feature-rich setup of BA51 with APLIC (8 interrupts), Debug module, Power management, PMP with 16 entries (16e), unified SPMP with 16 entries (16e), Memory debugger, CLINT, Triggers (8), Sstc, Trace, M-mode, S-mode, U-mode, and H (without virtual memory), A, C, F, H, I, M, N, Sspmp, Sstc, Zc, Zicntr, Zicsr, and Zifencei extensions.

From the synthesis results, we can see that adding virtualization support (H, PMP (16e) + unified SPMP (16e), APLIC, and SSTC extensions) does account for 5.1% of the area of the BA51-H FRC. But when we add 64KiB SRAM, the percentage of the area used by the added virtualization support drops to 1.4% of the whole area, which is an acceptable trade-off, especially if looking at the alternative of using multiple processing cores to obtain similar isolation guarantees. Furthermore, by using SPMP with 16 entries + unified SPMP with 16 entries instead of only SPMP with 32 entries, we only increased the design size by 490 gates.

Reference execution environment: In order to simplify the development process, we have first ex-



(a) High-level architecture of the BA51 core

(b) High-level architecture of the BA51-H core

Figure 6: Comparison of the BA51 and BA51-H core architectures

Table 1: The synthesis results of BA51 CC and BA51-H FRC.

Features	Gates	% of (#1)	% of (#2)	% of (#3)	% of (#4)
(#1) BA51 CC	25239	100.0	4.6	12.2	3.5
(#2) BA51 CC + 64KiB SRAM	549527	2177.3	100.0	266.2	75.2
(#3) BA51-H FRC	206430	817.9	37.6	100.0	28.3
(#4) BA51-H FRC + 64KiB SRAM	730718	2895.2	133.0	354.0	100.0
Hypervisor extension	7685	30.4	1.4	3.7	1.1
PMP (16e) + unified SPMP (16e)	51223	203.0	9.3	24.8	7.0
PMP (32e)	50733	201.0	9.2	24.6	6.9
APLIC	1403	5.6	0.3	0.7	0.2
Sstc	904	3.6	0.2	0.4	0.1
Zc	1287	5.1	0.2	0.6	0.2

tended the Spike simulator [32], considered a golden model of RISC-V specification, to set up a reference execution environment that is as close as possible to the extended BA51 core. Having a reference execution environment is useful as it allowed us to test the implementation of BA51-H against the reference environment and, at the same time, start porting the CROSSCON Hypervisor before the BA51-H was implemented. The Spike simulator already supported most of the extensions needed for the reference execution environment. For the initial environment, we have configured Spike to simulate a single 32-bit core with rv32imafch_pmp_sstc_zc_zicntr ISA with memory mapping disabled. We further extended the Spike simulator with the unified (two-stage) SPMP extension that allows the hypervisor to restrict access to the memory. This required adding the SPMP and vSPMP-related registers, copying and adjusting the logic of PMP extension, and taking into consideration when the core is in one of the virtualized modes. We have shared the code with the RISC-V community [33].

Unified SPMP: The SPMP is a memory protection unit (MPU) that allows isolation of M-mode and S-mode managed resources by restricting access to memory. The SPMP specification [34] is currently being extended so that it can also be used by the hypervisor (H-mode). The current proposal suggests two separate SPMP units that are controlled by the hypervisor: one that restricts access of the virtualized

modes (VU-mode and VS-mode) to memory and the other that restrict access of the U-mode to memory. As part of the CROSSCON project, we proposed a unified SPMP model that only uses one SPMP unit to restrict access to both virtualized and U-mode. We think that the unified SPMP model provides a cleaner design while reducing the number of unused registers if virtualization is not used. We have extended the Spike simulator with a reference implementation [33] of the unified SPMP extension, and furthermore, implemented the extension in hardware as part of the BA51-H core. This work was also presented as part of the RISC-V summit 2024 [35].

Verification: We verified the behavior of the newly added extensions through differential testing. This method involves running each test in two environments: the actual BA51-H design and an extended version of Spike, which serves as the reference execution environment. Any discrepancies between the two executions indicate potential issues that can then be identified and resolved. Furthermore, we have run a set of fuzzy generated tests to ensure that core functionality of BA51 remained intact. This was done in collaboration with Technical university of Darmstadt, which generated the tests. Additionally, we have formally verified the core behavior of Beyond's AHB interconnect using Symbiyosys [36], an open-source formal verification that allows SMT-based testing. We have described our formal verification effort in D2.4 [37]. Some of the verification activities (e.g. APLIC verification) are still in progress and plan to be finished in the following months.

4.1.5 Isolation of the interconnect using Perimeter guard

A *Perimeter guard (PG)* is a mechanism that allows one to (1) control which hardware modules can communicate with each other over the interconnect (on a granularity of a transfer), and furthermore, (2) allows the HW modules to be shared between different VMs / domains while preserving their isolation. A detailed description of PG is available in Chapter 7.2 of D2.3 [29], where in this chapter we highlight PG's functionality relevant for efficient cross hardware-software stack isolation.

(1.) Interconnect isolation: PG leverages several PG^{in} and PG^{out} modules to tag every transfer over the interconnect with the ID of the domain the transfer is coming from, called DID. This allows us to separate all HW modules connected to an interconnect into separate domains, where only HW modules that are in the same domain can talk to each other. This allows us to isolate all the transfers based on the domain they belong to. Where if the space of all DID's is large enough, we can put each HW module in its own domain, which allows us to control precisely which modules can communicate, basically getting the ability to securely exchange messages over the interconnect.

The idea of domains can be further extended to TEEs / VMs running on top of the CROSSCON Hypervisor and BA51-H. If we can propagate the ID of the VM to the interconnect and pass it to the interconnect as DID, we can guarantee isolation directly from the VM all the way to the HW module the VM is talking to. This allows us to treat VMs as if they are a part of a domain. Figure 7 shows a setup where we have a CROSSCON SoC with two domains, domain 1 marked with blue and domain 2 marked with red. VM1 is talking to the AES-GCM accelerator, and, because VM1 and AES-GCM are a part of the same domain, no software or HW module from the other domains can eavesdrop on the communication. Note that the AES-GCM accelerator is directly memory-mapped to the VM address space and does not require any additional driver from the software side; all the read and write instructions result directly in reads and writes on the interconnect that are transferred to the accelerator.

(2.) Sharing HW modules: Furthermore, PG can used to share hardware modules between different domains in a secure manner, by allowing the module to be switched from one domain to another in a way to prevent any unwanted information flows. This can be done by either "resetting" the module or by switching the module's context / state before granting access to a different VM / master. Resetting and switching the module's state prevents state-related information flows through the HW module, and furthermore, can be used to address timing-related attacks. PG supports several operation modes, which are selected during the integration into a SoC. For example, in the CROSSCON SoC, AES-GCM

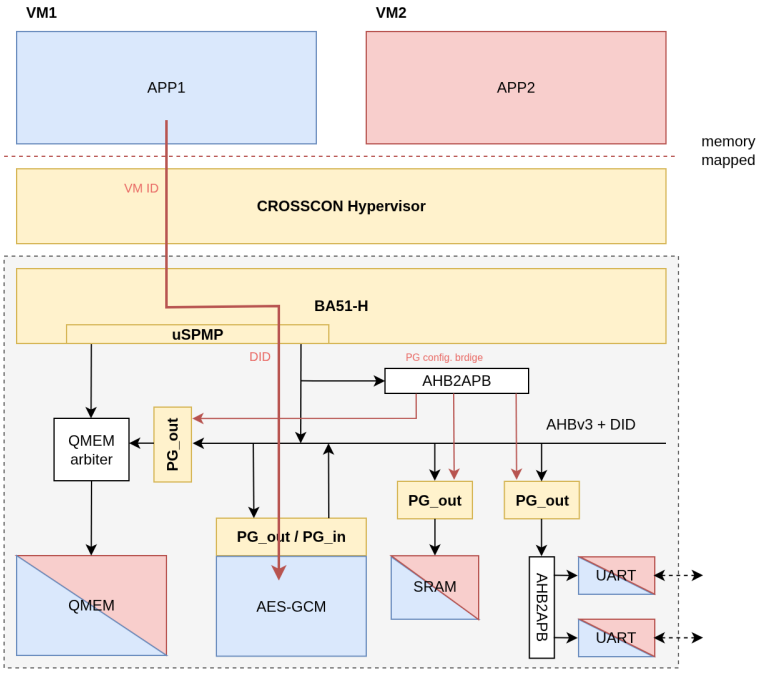


Figure 7: An example of how a read / write requests are propagated from the software running in a VM to the AES-GCM accelerator, where blue denotes resources that are a part of domain 1 and red denotes resources that are a part of domain 2.

supports context switching and is thus used with PG in Time-sharing with context switching operation mode.

Hardware resources: Table 2 shows the synthesis results of PG modules in different operation modes. We used the same synthesis and comparison approach as for the BA51-H core (Table 1). Furthermore, we have included the size of BA51 CC and BA51-H FRC setup for comparison. The PG^{out}_lock_release design refers to PG^{out} in Lock-release with reset arbitration mode, where PG^{out}_rst_addr_access design refers to PG^{out} in Restricted address space mode with 8 registers. Note that we did not include the synthesis result of PG in Time-sharing with context-switching mode as we only implemented such a mode as part of the AES-GCM accelerator, which synthesis results are visible in Table 6.

Table 2: The synthesis results of PG^{out} in Lock-release with reset arbitration mode (PG^{out}_lock_release) and PG^{out} in Restricted address space mode (PG^{out}_rst_addr_access)

Design	Gates	% of (#1)	% of (#2)
(#1) BA51 CC	25239	100.0	12.2
(#2) BA51-H FRC	206430	817.9	100.0
PG^{out}_lock_release	1397	5.5	0.7
PG^{out}_rst_addr_access	13438	53.2	6.5

The synthesis results show that the PG^{out}_rst_addr_access is approximately 10-times larger then PG^{out}_lock_release. We assume that the increase in size is mostly due to the range-related registers needed in PG^{out}_lock_release. Furthermore, we can see that PG^{out}_lock_release uses only 0.7% of BA51-H FRC area, where PG^{out}_rst_addr_access uses 6.5%.

4.1.6 AES-GCM accelerator with context-switching support

In this chapter, we describe how we have extended the AES-GCM hardware accelerator to support context switching so that it can be used by multiple domains through PG in Time-sharing with context-switching mode. Furthermore, we describe how the module was integrated into the CROSSCON SoC.

AES-GCM is a widely used authentication cipher based on AES [38] algorithm and Galois counter mode (GCM) [39]. It can be used to encrypt / decrypt messages and to check the messages authenticity through a tag calculated alongside the encryption / decryption. The encryption / decryption of a message and calculation of a tag can be performed in parallel, reducing the time needed to process a message. Internally AES-GCM uses the AES algorithm to develop a key-stream that is xor-ed with the message, 128-bit blocks at a time. This allows us to perform the same procedure to encrypt and decrypt a message, which reduces the amount of logic and state needed to process a message. This makes the AES-GCM algorithm an interesting candidate for adding context-switching support.

AES-GCM accelerator: For the initial AES-GCM implementation, we have used Beyond's AES-GCM accelerator that is build around a simple AES module that processes a single 128-bit block at the time. The module does not use pipelining, which means that a new block can be passed to the module when the module has finished processing the previous block. The module has a fairly standard architecture, where one clock cycle is required to perform a single round of AES algorithm. Besides that, the module needs 3 additional clock cycles to pre- and post-process a block, which makes the total number of cycles needed to process a block 13 (10 + 3) cycles for 128-bit keys and 17 clcyes (14 + 3) for 256-bit keys. This means that the module has a top processing speed of 0.9 Mbs/MHz for 128-bit keys and 0.7 Mbs/MHz for 256-bit keys.

Context-switching: We have extended the AES-GCM module with support for context switching that allows a context to be switched on block granularity. Figure 8 shows the basic idea behind switching. The module's interface was extended with additional signals that allow storing and restoring the context together with a signal (`cswitch_allowed`) that indicates when the context can be switched. This allows the module to switch between encrypting / decrypting modules from different contexts, interleaving the blocks until all the messages are encrypted / decrypted.

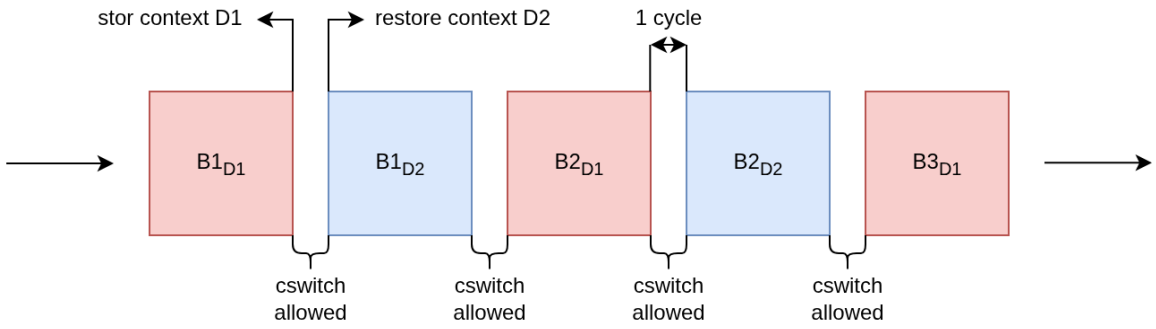


Figure 8: Interleaving of messages through context switching

The basic idea behind adding context switching support is relatively simple, as all we needed to do is to identify the modules state and allow an external module to modify it through storing and restoring. This can be easily done by identifying each register in the module and adding two sets of signals: one set that allows an external module to read the value of a register and the other to write a new value to the register, as shown in Figure 9.

But things get more complicated when we want to reduce the amount of state that we need to store for each context. We noticed that not every register needs to be stored if we limit ourselves to performing

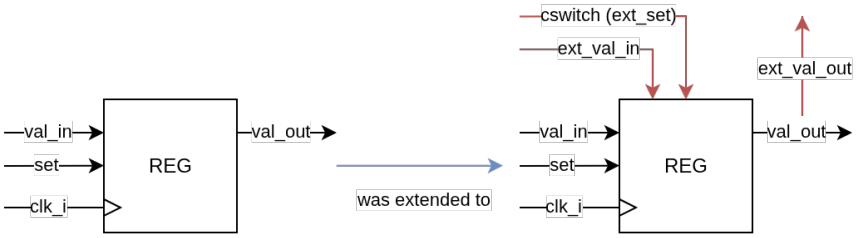


Figure 9: How registers were extended to allow an external module to read (*ext_val_out*) and write (*ext_val_in* and *cswitch*) the registers.

a context switch only when a module is in a specific state, e.g. the module has finished with decryption / encryption of a block. This allows us to ignore parts of the state needed to track and perform the steps of AES encryption / decryption and GCM calculation. Therefore, allowing us to only store the counters (and few other processing related values) needed to track which block is being processed instead of the intermediate values of a block computed on each cycle. As a general guideline, when adding context switching support to a module, we need to identify a state in the modules state machine that is regularly visited (i.e. is part of the state machines loop) and is as small as possible. In the case of our AES-GCM module, the state that we need to store for each context is 583-bits in size, which could probably be further reduced.

AES-GCM wrapper: We have integrated the extended AES-GCM accelerator into the CROSSCON SoC so that each domain can access the module directly without the need for software arbitration, as the arbitration is done directly in hardware. The AES-GCM accelerator is connected to the AHB interconnect through a PG compatible interface that implements Time-sharing with context-switching mode. The required logic is implemented as part of the AES-GCM wrapper which provides a simple interface that can be used by software and other masters on the interconnect, handles the arbitration between different domains based on DID, performs context-switching and drives the accelerator

AES-GCM software interface: The AES-GCM wrapper is memory-mapped directly to each VM without the need for software arbitration. This means that all read and write are passed directly to the wrapper. Each VM has access to a set of registers, listed in Table 3, that serves to control the encryption process and provide the cryptographic material. The wrapper, together with the accelerator, support the workflow shown in Figure 10. The accelerator can be used to perform encryption and decryption.

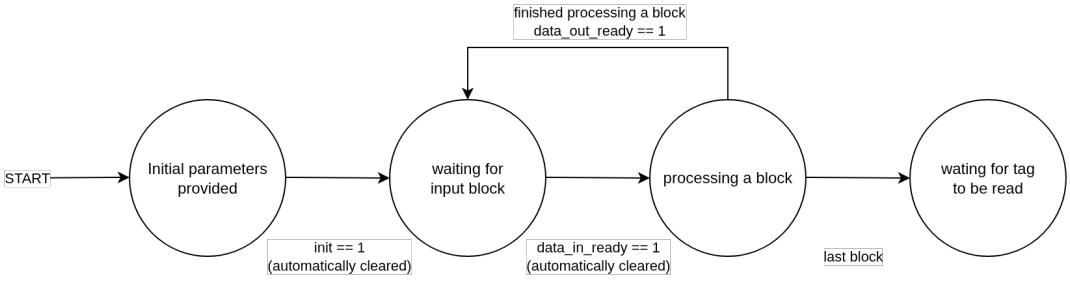


Figure 10: AES-GCM wrapper workflow

Note that in Table 3 and across this document we use a shorter name for the registers, where we replace AES_GCM at the beginning of the name with AG; for example, a short name for the AES_GCM_STATE_REG register is AG_STATE_REG.

Worflow:

1. Initially, the software needs to provide the initialization parameters (size of key, key, iv, etc.) to the accelerator and set the AG_STATE_INIT_FLAG flag in the AG_STATE_REG register. The accelerator

will perform the initialization and clear the flag when it is done. The initialization is currently performed in one clock cycle, thus software does not need to wait for the flag to be cleared.

2. When the initialization is done, the software can start providing additional data, block-by-block, by writing a block of additional data to the buffer pointed to by the address in AG_IN_D_ADDR_REG register, and setting the AG_STATE_IN_D_RDY_FLAG to note that the input data is ready. After the additional data is processed, the wrapper will clear the AG_IN_D_ADDR_REG and new block of additional data can be provided.

3. After all the blocks of additional data were provided, the software can provide the first block of data (plaintext / chiphertex). This is provided again block-by-block, writing a block to the buffer pointed to AG_IN_D_ADDR_REG address and setting the AG_STATE_IN_D_RDY_FLAG.

4. After processing a block of data (encrypted / decrypted), the wrapper will set the AG_STATE_OUT_D_RDY_FLAG to let the software know that the output block is available in the buffer pointed to by AG_OUT_D_ADDR_REG address. Once the software has read the output block, it can pass a new data block to the accelerator.

5. After all the blocks were processed, the wrapper will write the flag to the AG_OUT_D_ADDR_REG + 16 address, right after the last output block, and set the AG_STATE_TAG_RDY_FLAG flag to let the software know that the tag is ready.

An example of how the accelerator can be used in software can be found in CROSSCON's GitHub [40] repository under `examples/aes_gcm_acc_example/src/main.c`.

Note that the wrapper automatically retrieves and stores blocks of data from shared memory when data is available; it retrieves a 16-Byte block from the address range [AG_IN_D_ADDR_REG - AG_IN_D_ADDR_REG + 15] and, after it has finished processing the block, it returns the processed block to [AG_OUT_D_ADDR_REG, AG_OUT_D_ADDR_REG + 15]. Because the QMEM is protected by PG, we need to configure the QMEM's PG to allow the wrapper to access that part of the memory.

PSA Crypto API: The memory mapped interface provided by AES-GCM wrapper was designed to be compatible with single-part and multi-part interface provided by PSA Crypto API [11], allowing the accelerator to be integrated with PSA Crypto API compatible library.

Hardware vs software arbitration: When hardware resources are shared, we need to decide who and when will access the resources. That is usually done in software, for example, as a driver in the operating system (in S-mode) or the hypervisor (in H-mode), as it provides greater flexibility because the software can be updated. But doing arbitration in software can be slow and complicated. In the case of AES-GCM accelerator, we perform arbitration in hardware as part of the AES-GCM wrapper. This allows us to avoid adding additional code to the hypervisor or using a separate VM for the arbitration, reducing the size of the trusted code base. Furthermore, we can avoid additional synchronization and communication needed if multiple cores or masters try to access the accelerator. This makes the entire arbitration logic much simpler. In addition to that, the arbitration can be performed in a couple of clock cycles, avoiding the additional time overhead provided by software. If we were to perform the arbitration in software, doing context switching on block granularity would not make a lot of sense, as AES-GCM can encrypt / decrypt a block in approximately 33 cycles where in that time the processor can only perform a couple of 10 instructions. Furthermore, no configuration is needed to securely use the AES-GCM wrapper, as the arbitration mechanism is active right after reset.

AES-GCM wrapper architecture: Figure 11 shows a high-level architecture of the AES-GCM wrapper. The wrapper is composed from 4 sub-modules: a *driver* that drives the AES-GCM module and enforces the workflow, a *scheduler* that schedules the next domain whose block will be processed, a *register handler* that exposes the registers (Table 3) on the interconnect, and a *fetch_and_store* module that retrieves and stores blocks of data from shared memory (QMEM).

Table 3: The registers exposed by the AES-GCM wrapper, where AG stands for AES_GCM.

Register	Size (bits)	Addr[8:2] (dec)	Access	Description
AG_STATE_REG	32	0		Status registers. Table 4 the available flags.
AG_ENCRYPT_REG	32	2	RW	If set to 1, AES-GCM encrypts the block; otherwise decrypts it. Set by the program.
AG_KEYL_REG	32	3	RW	If set to 0x0, AES-GCM assumes 128-bit key length. If set to 0x1, AES-GMC assumes 192-bit key length. If set to 0x0, AES-GCM assumes 256-bit key length. Set by the program.
AG_KEY_REG[0-7]	32 x 8	3-10	RW	Registers holding the encryption key. Set by the programmer. In the case of 256-bit, all the registers are used to hold the key. In the case of 192-bit key, the first 7 registers are used to hold the key. In the case of 128-bit key, first 4 registers are used to hold the key.
AG_IV_REG[0-2]	32 x 3	11-13	RW	Registers holding the IV. Set by the programmer.
AG_A_LEN_REG	32	14	RW	Length of the additional data (i.e. additional data in the message) in Bytes.
AG_D_LEN_REG	32	15	RW	Length of the data (i.e. data part of the message) in Bytes.
AG_IN_D_ADDR_REG	32	16	RW	The address of the input block. Set by the programmer. The memory region the address is pointing to needs to be 128-bit long.
AG_OUT_D_ADDR_REG	32	17	RW	The address of the out block. Set by the programmer. The memory region the address is pointing to needs to be 256-bit long. The fist 128-bits are used by AES-GCM to return the processed block, where the next 128-bits are used to return the tag.

The driver acts as the primary coordinator of the modules. It manages the workflow progression and controls the AES-GCM module. It ensures that all required conditions are satisfied before transitioning to the next workflow state. It is the only module that communicates with the AES-GCM module and, when needed, also performs the context switching.

The register handler is responsible for managing the registers. It exposes them over the interconnect and makes them available to other modules in the wrapper. The handler keeps a separate set of registers for each domain, so that each domain has access to only one set of registers. The handler uses the DID to identify from which domain a read/write request comes from.

The scheduler determines which data block will be encrypted next using a Round-Robin algorithm. It only considers blocks that have already been provided, meaning that a domain needs to provide the data block before it can get time on the AES-GCM module.

Table 4: The flags of the AG_STATE_REG register

Flags	Bit's Index	Access	Description
AG_STATE_INIT_FLAG	0	RW	Set by the program when the initialization parameters are provided. Cleared by AES-GCM when initialization is complete.
AG_STATE_IN_D_RDY_FLAG	1	W-only	Set by the program when next input block is provided. Cleared when the input block is read by the AES-GCM wrapper.
AG_STATE_OUT_D_RDY_FLAG	2	R-only	Set by AES-GCM when it has finished processing a block and processed block is available. Cleared by AES-GCM when next input block is provided.
AG_STATE_TAG_RDY_FLAG	3	R-only	Set by AES-GCM when it has finished calculating the tag and tag is available.

The fetch_and_store is responsible for reading and storing blocks in shared memory. Once the schedule has chosen the next block to be processed, the fatch_and_store unit retrieves the block and passes it to the driver. Once the block was processed, the driver passes it back to fetch_and_store unit which stores it back to the shared memory.

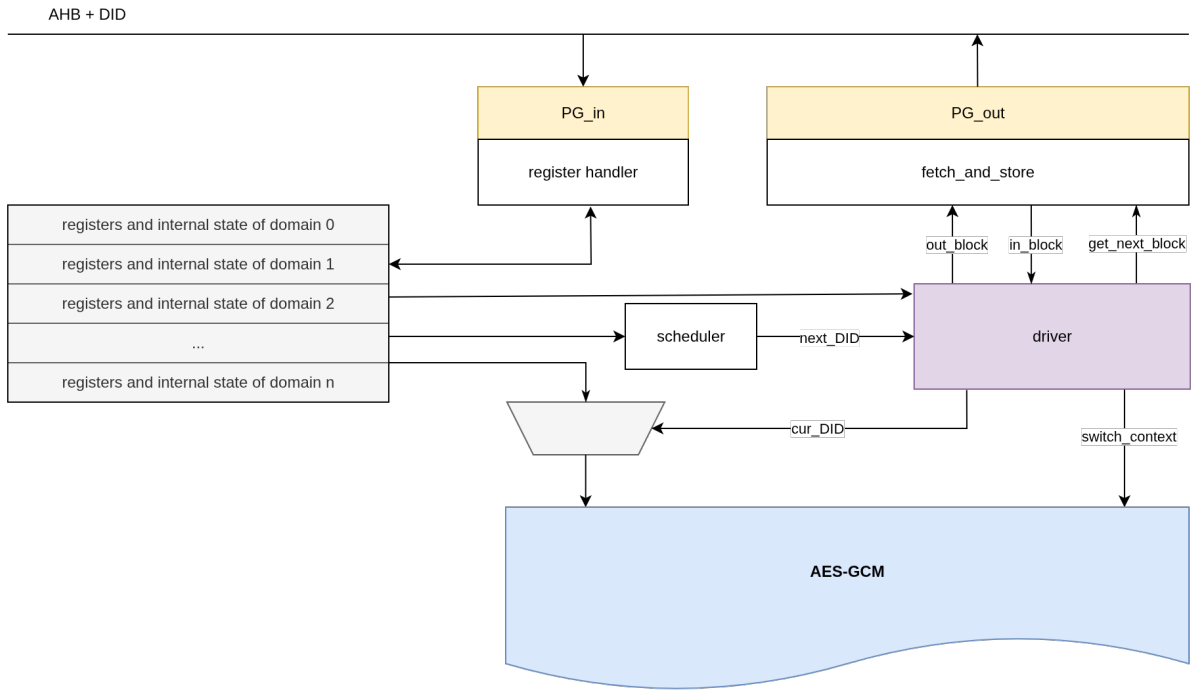


Figure 11: Architecture of AES-GCM wrapper

Performance: The best encryption / decryption speed that we were able to achieve with the AES-GCM accelerator through software interface, running on CROSSCON SoC on Arty-A7 with 25MH, is 12.20 Mbs for 256-bit key, which corresponds to 0.488 Mbs / MHz. This means that we are able to process a single 128-bit block of data in approximately 250 clock cycles. This also includes the software instructions needed to push the blocks of data to the accelerator, read back the result, and the logic needed to iterate through all the data. As we know that we are able to process a single block of data in 33 cycles

through the AES-GCM wrapper interface, it follows that we need approximately 217 cycles to perform the necessary software instructions, which is far from optimal. We could optimize the software part to improve performance but, as far as we can see, a better solution would involve adding additional logic to AES-GCM wrapper to automatically read the entire message from memory and writing back the result. This would allow us to avoid most of the software overhead. We plan to implement this solution as part of future work.

Table 5 shows the best speeds that we were able to achieve with extended AES-GCM accelerator encrypting / decrypting data from a single domain (without context-switching) with 256-bit key. The *(# 1) best* entry shows the speed that can be achieved in the best case [1], where we need only 14 cycles to encrypt / decrypt a block; one cycle for each operation of AES algorithm. The *(#2) AES-GCM HW* entry shows the speed achieved with unmodified AES-GCM modue, the *(#3) AES-GCM HW with cswitch* shows the speeds of AES-GCM module extended with context switching support, the *(#4) AES-GCM wrapper* shows the speeds that we are able to achieve through the AES-GCM wrapper using extended AES-GCM, and the *(#5) SW* shows the best speed that we were able to achieve using AES-GCM through software using memory mapped AES-GCM wrapper. We can see that pushing data to AES-GCM from software (#5), block-by-block, is 17-times slower then the (#1) best solution. Furthermore, if we compare the software and hardware usage of AES-GCM wrapper, using AES-GCM wrapper from software is 7.57-times slower than from hardware. This shows that execution of software is much slower than encryption / decryption of a block in hardware; makes removing software a good starting point for further optimization.

Table 5: The speed of the AES-GCM module used through different interfaces.

Interface	# cycles per block (256-bit key)	Mbs/MHz	× slower (#1)
(#1) best	14	8.719	1
(#2) AES-GCM HW	18	6.781	1.285
(#3) AES-GCM HW with cswitch	18	6.781	1.285
(#4) AES-GCM wrapper HW	33	3.699	2.357
(#5) SW	250	0.488	17.857

Note that the AES-GCM context switch takes only one cycle and that the AES-GCM wrapper always accounts for that cycle in 33 cycles it needs to process a block even if the context switching is not performed. This means that even if multiple domains use the module, the accelerator would process blocks at the same speed. But because the processed block belong to different domains, the domains would see a slowdown in the accelerator's performance equal to the number of domains using the module.

Hardware resources: Table 6 shows the synthesis results of basic AES-GCM accelerator design (*aes_gcm_acc*) and the AES-GCM accelerator design with context switching support (*aes_gcm_cs_acc*). We used the same synthesis and comparison approach as for the BA51-H core (Table 1). Furthermore, we have included the size of BA51 CC and BA51-H FRC setup to see how they compare to the size of the accelerators. Note that aes_gcm_acc is the initial AES-GCM accelerator design, the aes_gcm_cs_acc the accelerator design that we extended with context-switching, and the aes_gcm_cs_acc_wrap is the accelerator design with context-switching and wrapper.

From the synthesis results, we can see that adding the context-switching support to AES-GCM accelerator results in 11.4% increase in size. Furthermore, we obtain an additional increase 37% when adding the wrapper, getting an overall increase of 48.4%. Note that some of the wrapper's logic would be present even if the accelerator would not support the context switching, as we need to connect it to the interconnect, but we assume that most of the wrapper's logic comes from keeping multiple registers for different domains and storing different contexts. Therefore, by adding the context switching support and integrating the accelerator into the CROSSCON SoC, we get almost 50% increase in AES-GCM module's size, which represents approximately 26% of BA51-H FRC core. As far as we can see, this seems as

[1] When assuming an AES architecture where we process one block at the time and perform one step of AES algorithm per cycle.

Table 6: The synthesis results of AES-GCM accelerator (*aes_gcm_acc*), AES-GCM accelerator with context switching (*aes_gcm_cs_acc*), and AES-GCM accelerator with context switching and wrapper (*aes_gcm_cs_acc_wrap*).

Design	Gates	% of (#1)	% of (#2)	% of (#3)	% of (#4)	% of (#6)
(#1) BA51 CC	25239	100.0	12.2	23.5	21.1	15.8
(#2) BA51-H FRC	206430	817.9	100.0	192.0	172.4	129.4
(#3) aes_gcm_acc	107501	425.9	52.1	100.0	89.8	67.4
(#4) aes_gcm_cs_acc	119743	474.4	58.0	111.4	100.0	75.0
(#5) aes_gcm_cs_acc_wrap	159574	632.1	77.3	148.4	133.3	100.0

an acceptable tradeoff for situations where we do not need 2 accelerators all the time and the additional delay when processing a message is acceptable.

AES-GCM wrapper on the interconnect: Note that, because the AES-GCM wrapper is connected to the interconnect, other masters can also interact with the accelerator. This gives us additional flexibility, as we can use other HW modules, for example, DMA, to improve the encryption / decryption speed of messages (by avoiding software), but at the same time, requires us to restrict access to the wrapper on the interconnect level, which is done using PG.

Security analysis: The AES-GCM wrapper implements a PG with Time-sharing and context-switching interface where it uses a Round-robin algorithm for scheduling. Because the AES-GCM accelerator supports context switching, we are able to avoid any state related information flows as each domain has access to the accelerator only when it uses the context belonging to that domain; on each context-switch, wrapper loads the context belonging to the next domain before the next block is processed. Regarding timing attacks, because the scheduler does not grant time on the accelerator to every domain, but only the domains that provided a block to be processed, a domain is able to see when another domain is using the accelerator by timing how much time the accelerator needs to process a block that it provides. Furthermore, a domain can infer the size of key another domain uses as the amount of time the block is processed depends on the size of the key. Because the amount of time needed to process a block does not depend on the block's content, the key or the IV, we believe that the information leaked through timing cannot be used to learn anything relevant about other domains.

4.1.7 Demonstrator

A bitstream of the CROSSCON SoC with several examples can be found on CROSSCON's GitHub repository [40], where an example demonstrating all the features of the SoC can be found here [41].

The example contains the CROSSCON SoC with a CROSSCON Hypervisor that runs two guest VMs. Each guest VM is using the AES-GCM accelerator to encrypt and decrypt a block at the same time which causes the accelerator to process a block from one VM and then a block from the other. In order obtain the isolation between the VMs, we use a CROSSCON Hypervisors together with BA51-H to provide heart and memory isolation, we use PG to isolate the interconnect and we leverage AES-GCM context switching support to interleave the processing of the blocks. The example can be run on the Arty-A7 100T FPGA board following the instructions in the repository [41].

4.1.8 Summary

As we have shown, we can leverage BA51-H, PG and HW specific isolation mechanism, such as context-switching, to achieve an efficient cross hardware-software stack isolation, and furthermore, support an easy-to-use software interface with minimal time overhead. From the synthesis results, we can see that the addition of BA51-H and PG represents a minor increase in size of the SoC in comparison to the size of

SRAM, where AES-GCM accelerator size increased for almost 50% when we added support for context-switching and the logic needed to connect the accelerator to the interconnect. Overall, the increase in size is small but significant compared to the area used by other modules that are not SRAM. We argue that increase in size is reasonable, especially for applications where strong isolation is needed.

As part of our future effort, we intend to improve the maturity of all our solutions so that they are ready for production, and we plan to further support the inclusion of the suggested RISC-V extension into the RISC-V specification.

4.2 Hardware-assisted memory safety

Stack memory safety violations, such as buffer overflows, continue to pose a significant security challenge in software developed using memory-unsafe languages such as C and C++. Hardware advancements such as memory tagging schemes offer a promising foundation for enhancing memory safety by enabling the tagging of both pointers and memory objects. Unfortunately, existing tag-based approaches fail to protect pointer tags, which can be manipulated during pointer arithmetic operations that are common in system-level code, thereby rendering the security guarantees provided by current solutions questionable.

In this section we describe TAGShield, a runtime exploit mitigation mechanism that leverages the ARM Memory Tagging Extensions (MTE) to provide a persistent and deterministic defense against stack-based spatial memory errors, while preserving the integrity of pointer tags. TAGShield provides strong security guarantees through a transparent compile-time tagging scheme, complemented by a lightweight software instrumentation layer that enforces tag integrity.

4.2.1 Overview

Despite the long-standing recognition of the threat posed by stack-based spatial memory errors, this class of bugs remains a critical problem in software security. To balance security and performance, hardware-assisted memory safety features in modern architectures offer a promising solution to mitigate memory corruption vulnerabilities. Among these approaches, ARM's Memory Tagging Extensions (MTE) has emerged as a significant advancement in addressing spatial errors [42]. The MTE mechanism associates to every memory location and pointer a 4-bit tag, and enforces a hardware check to ensure that pointer dereferences only occur when the pointer and memory tags match.

While MTE-based solutions [43, 44, 45, 46] reduce the runtime overhead compared to software-only approaches [47, 48, 49, 50], they come with significant limitations. Most defenses leveraging MTE rely on hardware-generated random tags, which offer only probabilistic security guarantees [43, 44]. The low entropy of these tags exposes applications to brute-force attacks, while tag collisions between neighboring memory objects leave systems vulnerable to contiguous overflows.

In the context of the CROSSCON project we developed TAGShield, a runtime exploit mitigation mechanism designed to defend against stack-based spatial memory errors. TAGShield combines memory tagging with selective software instrumentation to address the limitations of previous techniques, providing deterministic protection while preserving the integrity of pointer tags. This approach effectively mitigates all pointer corruption attacks, including malicious arithmetic manipulations, providing a robust defense against stack-based spatial memory errors.

The core of TAGShield ’s design is based on two key principles:

- Transparent Memory Coloring Scheme: TAGShield employs an efficient memory coloring mechanism to selectively tag unsafe pointers and objects, ensuring spatial isolation and preventing memory collisions between adjacent regions.

- Lightweight Software Instrumentation: TAGShield introduces a software layer that persistently enforces the integrity of pointer tags, making them immutable and resistant to corruption. This layer ensures that adversarial attempts to manipulate tags are thwarted.

TAGShield strikes an optimal balance between performance and security through a selective protection strategy that focuses on vulnerable stack allocations. It also leverages compile-time tag generation to significantly reduce runtime overhead while maintaining robust security guarantees. Notably, TAGShield outperforms all state-of-the-art bounds-checking mechanisms with comparable security guarantees. Furthermore, its design does not introduces compatibility issues, minimizing barriers to adoption in real-world applications.

4.2.2 Adversary Model

We assume a software-based attacker who can exploit any stack-based memory vulnerability to corrupt memory. In particular, the adversary is assumed to have access to the target application's source code, binary, and compiler information of the target application, enabling detailed analysis of the target's memory layout and behavior. However, the attacker cannot tamper with the source code, binary, or compiler prior to deploying the target binary. Additionally, the adversary may attempt speculative execution attacks to infer memory tags [45], but cannot directly modify MTE tags or alter the MTE configuration, as the adversary operates in user mode and lacks the necessary privileges.

4.2.3 Design

Despite being an hardware extension, MTE can still incur in non-negligible overhead due to the cost of tagging memory [51, 45]. To reduce the overhead without compromising TAGShield security guarantees, we (i) carefully select the vulnerable stack allocations to protect and (ii) employ a static colouring schema.

Isolation of Safe Objects. The more allocations need protection, the more overhead MTE incurs for colouring memory and performing tag checks. For this reason, TAGShield selectively instruments only unsafe objects. Objects and their associated pointers deemed safe are efficiently isolated by retaining their default tag (typically zero). In contrast, unsafe objects and their associated pointers are assigned a non-zero tag chosen from the remaining values in the 4-bit tag space. TAGShield not only ensures that unsafe allocations are fully isolated from safe ones by preventing the forgery of the default tag, but also guarantees that unsafe allocations cannot forge each other's tags.

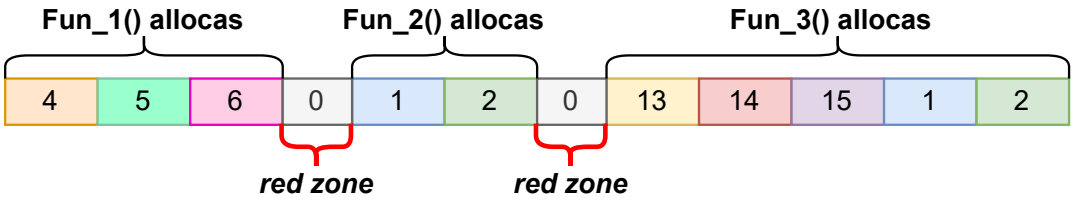


Figure 12: TAGShield's coloring schema. Three nested function cycle the available colors starting from a random tag. Between each function, a 0-tagged red-zone.

Deterministic Tagging of Unsafe Objects. The core principle behind TAGShield's coloring scheme is simple: whenever a function is invoked, we explicitly assign a unique tag to each of its stack allocations and propagate this tag to the corresponding pointer. We use statically computed tags determined at compile time and hard-code them within the application. Figure 12 illustrates TAGShield's scheme.

In the function prologue, we assign tags to all unsafe allocations by cycling through the available tag values, ensuring that adjacent allocations receive different tags. The first tag in each function is chosen randomly from the set of non-zero tags, after which subsequent allocations are assigned tags by cycling through the remaining values. These tags are cleared in the function epilogue when the corresponding stack frame is removed.

To further reduce the risk of tag collisions across function boundaries, TAGShield's instrumentation pass inserts a 16-byte padding granule (i.e., a red zone) after each stack frame. This prevents the first tag assigned in a new stack frame from inadvertently matching the last tag of the previous frame. This red zone is assigned the default tag, ensuring that no unsafe pointer can access it.

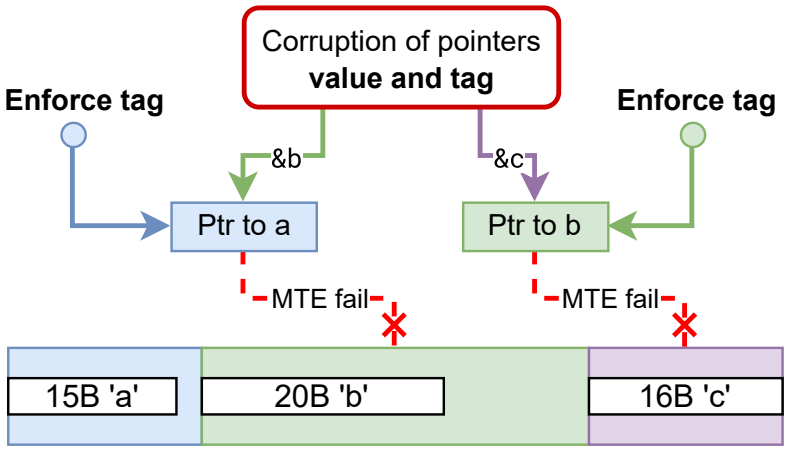


Figure 13: Example of attacker corrupting both tag and value of the pointers with TAGShield enabled. This restores the tag before the pointers are used.

Enforcing Integrity of Tags. Pointer arithmetic, which is common in programming constructs, not only allows adversaries to manipulate addresses to target allocations with matching tags but also enables them to forge tags themselves, as MTE does not protect tags. TAGShield enforces that a pointer can only be dereferenced if its tag matches the tag assigned by the employed coloring scheme at compile-time, regardless of whether the embedded tag has been corrupted. Figure 13 illustrates the tag integrity mechanism of TAGShield.

Upon accessing a tagged pointer, TAGShield ensures that its tag is authentic. For every unsafe alloca instruction that generates a root level pointer, TAGShield's plugin in the compiler toolchain generates and embeds a distinct tag. The integrity of this tag is safeguarded by hard-coding it in the preamble of the corresponding stack frame and enforcing it with every access. In other words, whenever the pointer is accessed, we reset its tag to the authentic one computed at compile-time, ensuring that it remains persistent throughout the pointer's lifecycle.

Non-root pointers, however, introduce additional challenges since their authentic tags may not be known at compile-time. Interestingly, since MTE tags are embedded in pointers without integrity guarantees, altering a pointer's content automatically updates its tag to match the source pointer. Nevertheless, TAGShield must track these tag changes to ensure tag integrity at runtime. Specifically, at any point during execution, TAGShield must determine the correct tag for non-root level pointers. We refer to the collection of all non-root level pointer tags as TAGShield *state*. TAGShield maintains this state using one reserved register (`TAG_REG`) within its software instrumentation layer.

For each non-root pointer in a function, TAGShield reserves 4 bits in `TAG_REG`. Whenever a non-root pointer is assigned a new address, its corresponding `TAG_REG` entry is updated with the tag of the source pointer. Since AArch64 registers are 64-bit wide, TAGShield can encode up to 16 non-root pointers per function. Our analysis shows that this is sufficient for many real-world applications, as functions rarely

require tracking more than 16 non-root pointers. In cases where a function makes a call to another function that also utilizes `TAG_REG`, the register is spilled to memory in the function prologue of the callee and restored in the epilogue, ensuring consistent tracking across function boundaries.

By tracking non-root pointer tags, TAGShield instruments each non-root dereference to enforce the authentic (aka correct) tag, retrieving it from `TAG_REG`. During compilation, TAGShield determines the location of each non-root pointer's tag in `TAG_REG`, enabling efficient runtime enforcement.

4.2.4 Implementation

To implement TAGShield, we leverage LLVM 18 [52], a widely used compiler toolchain in both industry and academia, known for its modularity and extensibility. TAGShield's implementation consists of two major LLVM components: a front-end pass and a back-end module. The front-end pass operates at the function-level of LLVM IR of a target application to detect pointer aliases and instrument it with custom intrinsics—special instructions that manage both static and dynamic tags at runtime. The front-end is implemented using LLVM's new pass manager and consists of approximately 1500 lines of code (LoC). To perform various analyses, it leverages several existing LLVM passes, including `StackSafetyAnalysisPass`, `AAResultsWrapperPass`, `StackSafetyGlobalInfoWrapperPass`, `TargetLibraryInfoWrapperPass`, and `MemoryDependenceWrapperPass`.

Although the LLVM IR is architecture-independent, TAGShield's implementation is tailored for AArch64, requiring architecture-specific instrumentation. The IR generated by the front-end must be translated into AArch64 assembly, which is handled by TAGShield 's back-end module. This module extends the AArch64 LLVM back-end with custom lowering mechanisms, converting TAGShield 's intrinsics into efficient ARM assembly instructions. While our implementation includes several optimizations to generate performance-efficient instruction sequences, most intrinsics are lowered using just two key ARM instructions: `movk` and `bfm`. The former only updates 16-bit segments of a register, while the latter moves arbitrary bit subsequences from one register to another.

5 Secure FPGA Provisioning

Field-Programmable Gate Arrays (FPGAs) have become integral components in modern computing environments due to their adaptability and versatility. We explore the advancements and challenges in FPGA virtualization, focusing on the partitioning of physical FPGAs into multiple virtual FPGA (vFPGA) for efficient resource utilization. While FPGAs are increasingly deployed in multi-tenant systems, today's platforms lack a trustworthy way to virtualize the fabric and to protect Intellectual Property (IP) across the full lifecycle of partial reconfiguration. Our objective is to deliver comprehensive and dynamic IP protection solutions on shared FPGA resources beyond existing work. We present a TEE-mediated control plane that partitions a physical FPGA device into multiple vFPGAs, enforces per-client authorization, and provisions accelerators using encrypted, signed bitstream packages bound to specific clients, while remaining GP-compliant. In the following sections, we provide insights into our approach towards Secure FPGA Provisioning with simultaneous clients providing strict isolation guarantees, discussing the architecture, implementation, and evaluation strategies.

5.1 Background on FPGAs

FPGAs are integrated circuits empowering users to directly program/configure and execute their hardware designs. These platforms offer the flexibility to reconfigure the hardware fabric with diverse hardware designs and various requirements. They consist of three major components: configurable logic elements, configurable interconnects, and Input/Output (I/O) blocks, which provide off-chip connections. A configurable logic element comprises a set of locally interconnected Look-Up Table (LUTs) and flip-flops. Configurable logic blocks are connected together or to I/O blocks through programmable routing interconnects. Other components of FPGAs are Block Random Access Memory (BRAM) blocks and Digital Signal Processor (DSP) blocks for high-performance DSP applications.

FPGA's development flow encompasses the sequential stages necessary for converting an abstract circuit description into a fully operational circuit deployed on the designated FPGA platform. During synthesis, the abstract circuit description is translated into a gate-level netlist representation. Following synthesis, the place-and-route phase involves mapping the generated netlist onto the resources of the target FPGA, including LUTs, flip-flops, BRAMs, and DSP cores. This step is critical for defining routing resources that establish connections between allocated resources while adhering to timing constraints such as operating frequency. Developers can influence this process by specifying placement constraints, such as confining the design to a specific region within the FPGA or dictating routing paths through designated channels. Subsequently, the generation of the binary file, which configures the target FPGA, occurs in the bitstream generation phase. An inherent or hardwired configuration engine embedded within the FPGA processes the incoming bitstream, utilizing it to program the configuration memory. This memory then dictates the behavior of the programmable logic elements and interconnections within the FPGA, following the predetermined hardware design.

Partial Reconfiguration (PR) introduces the capability for dynamic reconfiguration of regions of the FPGA while the remainder of the logic continues to function seamlessly. This approach involves partitioning the FPGA into a static region and one or more partially reconfigurable regions that can be configured at runtime. In the following, we discuss how this feature can be leveraged to enable FPGA virtualization. More background information on FPGA architectures and features can be found in [53].

5.1.1 FPGA Virtualization

In the fast-paced evolution of computing environments, including cloud and traditional setups, incorporating virtualization technologies is crucial. These technologies offer advantages such as flexibility, autonomy, isolation, and efficient resource utilization. Expanding upon traditional virtualization, the concept of FPGA virtualization has emerged[54], which involves abstracting away FPGA's complexities and simplifying the interaction interface. However, the distinctive characteristics of FPGAs, including their heterogeneous fabric, various execution models, and vendor-dependent logic, pose specific challenges and necessitate tailored requirements for successful virtualization implementation.

Initially, FPGA virtualization focused on temporal FPGA multiplexing to swap in and out parts of larger designs when device capacity was limited. Temporal sharing/multiplexing aligns more closely with Central Processing Unit (CPU) and I/O virtualization concepts. Nowadays, FPGAs can support spatial sharing. This concept aims to optimize resource utilization and maximize the return on investment of FPGA resources by leveraging the capacity for multiple hardware circuits to coexist on the FPGA fabric.

Over the years, FPGA virtualization has advanced significantly [55, 56, 57, 58]. Most FPGA virtualization schemes aim to abstract hardware-specific details and represent the FPGA and its components as a resource pool that a hypervisor can actively manage.

The subsequent challenge in FPGA virtualization revolves around determining the level of granularity at which FPGA primitives are abstracted into a resource pool for users. Outlining FPGA resources into fine-grain entities such as registers, programmable look-up tables, I/O blocks, and memory blocks for hypervisor management is intricate, mainly due to the diverse FPGAs' architectures and the reliance on the FPGA's specific fabric and layout details in current FPGA-based development. These details are crucial for the configuration toolchain to generate a compatible FPGA bitstream, thus making hardware-independent FPGA bitstream generation and configuration an ongoing research area.

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

In addition to the vFPGAs, a dedicated area within the FPGA is allocated for the static Programmable Logic (PL). This static PL is crucial in setting up the vFPGA partitions and supplying clock signals. The static PL can also help facilitate I/O for vFPGAs through PCIe or AXI interfaces, depending on whether the FPGA is an external hardware component or integrated on-chip with the processor. Optionally supporting the configuration of vFPGA resources via an internal interface. Optionally, the static PL can also support the configuration of vFPGA resources via an internal interface.

5.1.2 IP Protection

Partial bitstreams, which embed Hardware IP (HW IP) designs, are used to configure the vFPGAs. Ensuring the security of these partial bitstreams is paramount during storage, throughout the configuration process on the FPGA, and during deployment. The protection of FPGA bitstreams and associated IPs has been explored in existing literature [59] and addressed by FPGA vendors. Leading FPGA vendors like Intel and AMD offer solutions for bitstream protection, integrating cryptographic engines within their FPGA chips. The owner of an FPGA can program their cryptographic keys onto the FPGA, enabling decryption and integrity verification of the loaded bitstreams. However, our focus revolves around scenarios where different users supply their hardware designs for execution on the FPGA.

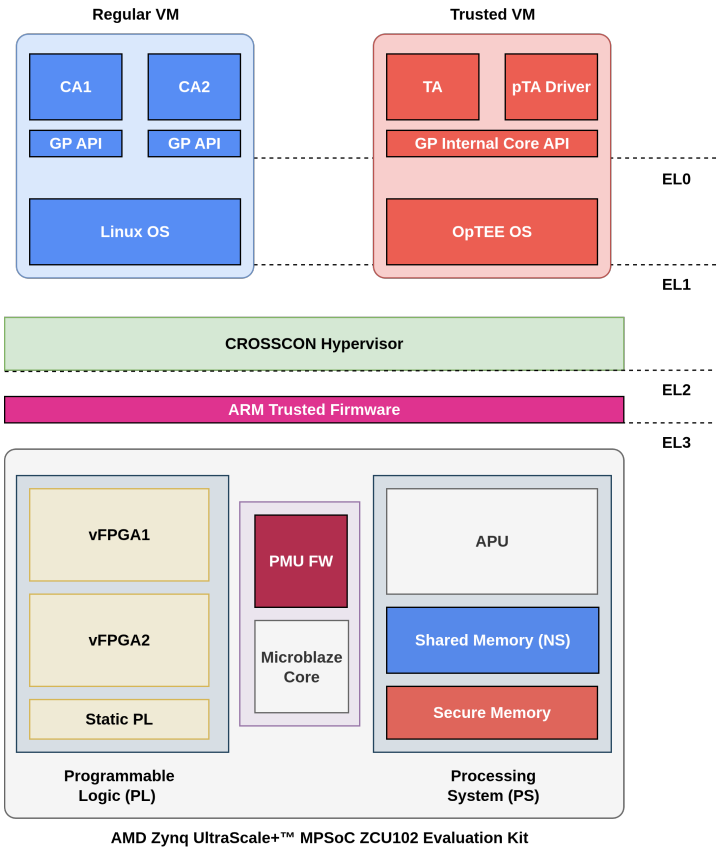


Figure 14: The architectural components relevant to T4.3.

To address this challenge, use a TA that runs in TEE and exclusively controls FPGA services. The partial bitstream can be decrypted, verified, and sent to one of the external configuration ports of the configuration engine on the FPGA over one of the available interfaces to the FPGA, e.g., JTAG, PCIe, etc. Once the configuration is completed, the hardware design/accelerator is made available to the requesting trusted application. The external configuration ports must be protected from unauthorised access and be fully under the exclusive control of the configuring enclave. To enable IP protection, a key exchange protocol between the application requesting the FPGA services and the TA can be used to share a secret key. The secret key is utilized to encrypt the partial bitstream.

5.2 Architecture

Figure 14 presents the overall architecture used for Secure FPGA Provisioning. The following is a list of the key components in the stack.

- **Client Applications (CAs).** Client applications CA1 and CA2 are executables that run on top Linux OS. The CAs act as the tenants of the shared FPGA. Each of these applications is controlled by clients who want to configure a partial bitstream on the FPGA for acceleration. They invoke the Trusted Application (TA) with an encrypted and signed package of their partial bitstreams for reconfiguration.
- **Trusted Application (TA).** The Trusted Application (TA) runs inside a trusted VM with OPTEE OS. It authenticates each package sent from a CA and decrypts it inside secure RAM. Since the TA cannot access the hardware directly, it hands off the partial bitstream to the pseudo TA (pTA) driver. The responsibilities of the TA are the following.
- **Pseudo TA (pTA).** The pseudo TA acts as an FPGA-reconfiguration driver in the Trusted VM. It receives

the partial bitstream from the TA, cleans caches, and issues a single Secure Monitor Call (SMC) that starts the hardware load sequence.

- **Regular VM.** This is a Normal world guest OS that hosts the CAs. It provides filesystem and driver support but has no direct access to secure memory or FPGA control.
- **Trusted VM.** This is the Secure world operating environment. It runs both the user TA and the core pTA, exposes the GP APIs, and owns the isolated secure DRAM region.
- **Hypv.** The CROSSCON Hypervisor is an Exception Level 3 (EL2) supervisor that the regular and trusted VMs, maintains second-stage page tables, and enforces isolation between the Linux and OP-TEE worlds. It also catches and transparently forwards the reconfiguration SMC from the pTA to the ARM Trusted Firmware.
- **ARM Trusted Firmware (TF-A).** This is the Highest privilege firmware (EL3). It receives reconfiguration SMC from the TA, checks buffer bounds, and forwards the bitstream to the platform management firmware for configuration. Only the TA is allowed to invoke this configuration path.
- **AMD Zynq Ultrascale MPSoC ZCU 102 Board (ZCU 102 Board).** This board is used as the underlying hardware for implementation. Internally, the Board can be split into two key components:
  - **Processing System (PS).** It hosts the APU cores, DDR controller, and the PCAP interface that streams configuration data from software into the FPGA fabric.
    - **Application Processing Unit (APU).** Quad-core ARM Cortex-A53 cluster.
    - **Shared Memory (Non-Secure (NS)).** Non-secure memory region in RAM is visible to both the VMs. It is managed by the hypervisor, and the key contents are always expected to be encrypted.
    - **Secure Memory.** Protected memory slice mapped only into secure-world address space and Exception Level 3 (EL3) privilege level TF-A.
  - **Programmable Logic (PL).** This is the programmable FPGA fabric on the board. In our implementation, we have a static component in the PL to manage the flow of information within the FPGA, ensuring smooth communication between the FPGA fabric and external devices. We have two reconfigurable partitions, vFPGA1 and vFPGA2, that can be reconfigured dynamically with partial bitstreams.
    - **Static PL.** Static component in the PL to generate clock, initialize internconnects, and define the reconfigurable partitions.
    - **vFPGAs.** Two floor-planned reconfigurable partitions that only accept partial bitstreams delivered by the PMU.
  - **Microblaze Core and Platform Management Unit (PMU).** The Microblaze core is an independent microcontroller running the Platform Management Unit Firmware (PMU FW). It configures the partial bitstream into a reconfigurable partition in the PL when instructed by the ARM Trusted Firmware. Only the TF-A can invoke this configuration path.

5.2.1 Responsibilities and Capabilities

Table 7 presents the responsibilities and capabilities of each of the architectural components seen in Figure 14.

Table 7: Key Components of the architecture along with their responsibilities and capabilities.

Component	Responsibilities	Capabilities
CAs	- Assemble and encrypt partial bitstreams. - Request different trusted services offered by the TA.	- Full access to the normal-world file system. - Can invoke GP APIs.
TA	- Authenticate / decrypt bitstreams. - Track vFPGA ownership. - Expose trusted services to CAs.	- Access to secure memory, cryptographic HW/TEE services. - Can invoke sessions to pTA.
pTA	- Bridge between TA and TF-A. - Invoke calls to TF-A.	- Runs in the kernel context of OP-TEE. - Access to secure memory.
Hypv.	- Partition PS resources between Linux VM and OP-TEE VM. - Memory management. Bridge between TA and TF-A.	- EL2 privilege over both VMs. - Can catch requests to TF-A.
TF-A	- Handle invocation from pTA. - Run at EL3; decide if reconfiguration is permitted. - Call PMU services through PMU-FW wrapper.	- Full secure-monitor access; can reconfigure PL via PMU. - Code is ROM-resident; updates require re-flash.
PMU FW	- Drive Xilinx ICAP/PCAP to load partial bitstream.	- Runs on a dedicated MicroBlaze core in the PMU. - Full control over platform-management registers.

5.3 Assumptions and Threat Model

In general, the following assets can be identified on FPGAs:

- Hardware Designs (HW IP): Hardware designs to be configured on the FPGA are provided in the form of FPGA vendor-specific configuration files. These designs may be identified as Intellectual property (IP) of their owners. Thus, the security of these hardware designs must be protected.
- Input Data and Results (Data): data processed and produced by the hardware design running on the FPGA.

Note that HW IP can be either:

- Trusted Hardware Designs provided by the FPGA vendor or a reputable 3rd-party IP vendor.
- Untrusted Hardware Designs: users may supply their own hardware designs or get them from 3rd-party IP vendors.

Existing research [60, 61, 62, 63, 64, 65] has highlighted potential security risks in clients sharing FPGAs-as-a-service. Therefore, ensuring the trustworthiness of hardware designs is paramount to safeguarding against these potential security risks. Nevertheless, we consider physical attacks on the An FPGA that requires direct physical access or close proximity to mount probes on or near the device to measure physical aspects of the device is out of scope. Our objective is to deliver comprehensive and dynamic IP protection solutions on shared FPGA resources beyond existing work. This includes safeguarding IP designs not only during the provisioning and configuration stages, but also post-configuration by implementing fine-grained access control mechanisms for the deployed accelerators.

5.4 Trusted Services

In this section, we present the *trusted services* that include all the functionality that is exposed by the secure world to unprivileged clients. Details of the trusted services are presented in D3.3. Conceptually, the trusted services layer sits above the cryptographic primitives and below any application-specific logic in the Linux VM. We look at the three key trusted services relevant to Secure FPGA Provisioning: *vFPGA Management Service*, the *Client Registry Service*, and the *vFPGA Configuration Service*. Each service is implemented primarily in the TA and pTA, while the CROSSCON hypervisor and TF-A contribute transit and policy enforcement only where strictly necessary.

5.4.1 vFPGA Management Service

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

5.4.1.1 vFPGA Status Check

- **Trigger:** The status check path in vFPGA Management Service can be initiated directly by the client application.
- **Parameters:** Status check does not require any parameters; the invocation presents the state of both the vFPGAs.
- **Result:** If successful, the service results in a string with the state of two vFPGAs, accessible to the caller CA.
- **Pre-conditions for success:** The status check has minimal authentication done by the TA. The only point of failure is a malformed invocation by the TA.

5.4.1.2 vFPGA Status Update

- **Trigger:** The vFPGA status update path is *not* invoked directly by a client but automatically by the TA when the state of a vFPGA changes from free to occupied or vice-versa. This happens in two situations: (i) immediately after a successful `vFPGA configuration` and (ii) after a successful `vFPGA free`.
- **Parameters:** None. The TA operates purely on its vFPGA state list present in the secure memory.
- **Result:**
  - To avoid race conditions between different CAs invoking an update, the TA acquires a spin-lock before modifying the status.
  - In case of a vFPGA configuration, the TA copies the caller CA's identifier to the corresponding vFPGA state in secure memory, marking the slot as *occupied*.
  - In case a CA frees the vFPGA, the TA zero-fills the state of the corresponding vFPGA in the secure memory, marking it *free*.
  - The modification becomes visible to any subsequent `status` query through the vFPGA management service.
  - Once the modification is complete, the CA releases the spin lock.
- **Pre-conditions for success:**

a) For the transition from *free to occupied*, the enclosing `configure` command must have completed successfully.

b) For the transition from occupied to free, the caller CA's identifier must match the stored owner entry for the corresponding vFPGA.

5.4.2 vFPGA Client Registry Service

Includes key exchange and mapping the UUID with the public key of the clients. Only once for a CA. Without doing this, no configuration is possible.

- **Trigger:** A CA that is unknown to the TA can request this service to register itself with the TA.
- **Parameters:**
  a) CA's public key to be used for authentication in the vFPGA configuration service by the TA.
  b) a buffer to receive the TA's public key.
  c) CA's identifier.
- **Result:** The TA returns its own public key to the CA. For unknown clients, a new entry is created in the registry with the CA's identifier and public key. To avoid race conditions between different CAs invoking the client registry, the TA acquires a spin-lock before adding a new client.
- **Pre-conditions for success:**
  a) The parameter should be formatted correctly.
  b) There should be enough space in the shared memory for the TA's public key.
  c) There should be enough space in the secure memory to add the entry in the client registry.

5.4.3 vFPGA Configuration Service

Configuration is the only service that traverses all the privilege levels.

- **Trigger:** A known CA (now with an entry in the client registry) can invoke the vFPGA configuration service to configure a partial bitstream on a free vFPGA.
- **Parameters:**
  a) A package with an encrypted partial bitstream, signature, wrapped key for encryption, and necessary metadata.
  b) Scalar value $\in \{1, 2\}$ selecting the vFPGA.
- **Result:** On success, the partial bitstream is loaded into the matching reconfigurable partition. The TA returns a successful acknowledgement to the CA, and the status update path of the vFPGA management service is called to mark the CA as the owner of the corresponding vFPGA.
- **Pre-conditions for success:**
  a) The CA is known to the client registry.
  b) Requested vFPGA slot is valid and currently free.
  c) CA passes the ownership check of the package.
  d) Package passes the expected sanity checks.
  e) Rivest-Shamir-Adleman (RSA) signature over the wrapped encryption key verifies successfully.

f) Encryption authentication tag verifies successfully.

g) Secure memory has sufficient free space for the encrypted package and the partial bitstream.

5.4.4 vFPGA Deallocation Service

- **Trigger:** The CA invokes the vFPGA deallocation service when it no longer needs exclusive access to a vFPGA slot.
- **Parameters:**
  a) any package previously produced by the CA (the TA uses only it to verify the caller is authentic).
  b) Scalar value $\in \{1, 2\}$ identifying the vFPGA slot to be released.
- **Result:** The corresponding entry in the vFPGA state maintained in the secure memory is cleared, and an optional dummy bitstream is configured on the corresponding vFPGA. Another CA or the same CA again may now claim the vFPGA.
- **Pre-conditions for success:**
  a) Package is well-formed.
  b) vFPGA-ID is valid.
  c) The CA passes the ownership verification of the package.

5.5 Workflow

5.5.1 vFPGA Status Check

The vFPGA status check sequence is kicked off when the CA running in the Linux VM calls

```
./ca_executable status
```

which does not require any additional command line parameters. Upon receiving the status check request, the TA works in two phases.

5.5.1.1 Phase1: Parameter sanity.

The TA verifies if the parameters used for invocation match the expected order. Mismatch results in a `TEE_ERROR_BAD_PARAMETERS` error, and the request is rejected.

5.5.1.2 Phase 2: State formatting.

In this phase, the TA performs the following steps.

1. Inspect the vFPGA owner list maintained across sessions in the secure memory.
2. Compose two human-readable lines:

   ```
   "vFPGA1: free|occupied\nvFPGA2: free|occupied\n"
   ```

   and share it back with the CA.

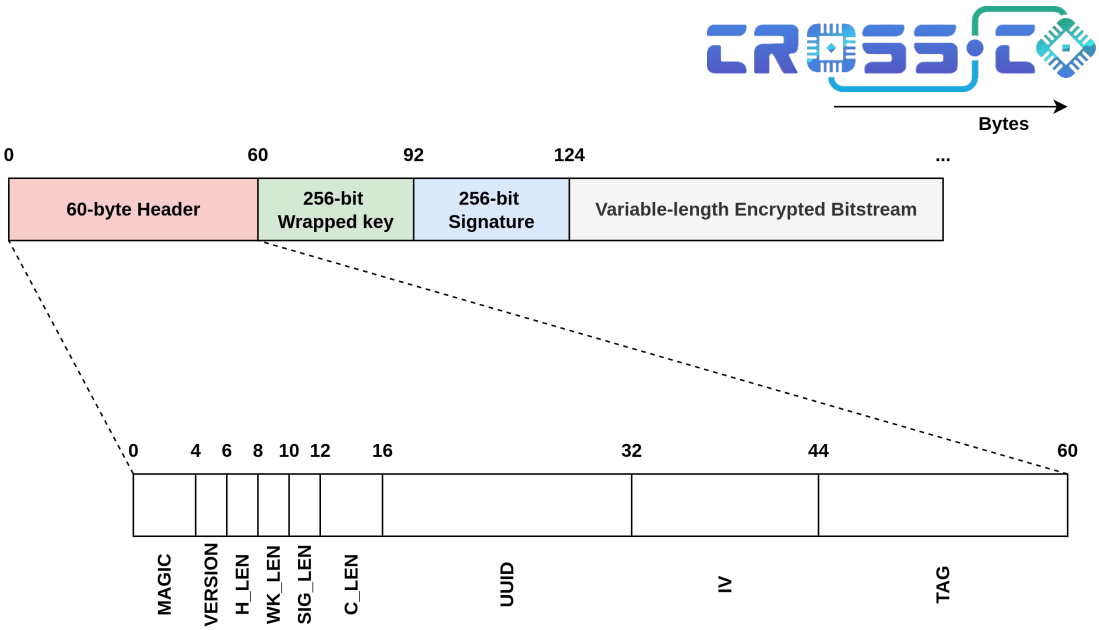


Figure 15: Organization of the package sent by the client application to request vFPGA configuration.

5.5.1.3 Successful Outcome

The CA prints the returned ASCII. This service involves no cryptography or privileged calls.

5.5.2 Key Exchange

The key exchange sequence is kicked off when the CA running in the Linux VM calls

```
./ca_executable get-key <ca_pub_key_blob.bin> <ta_pub_key_blob.der>
```

where `<ca_pub_key_blob.bin>` is CA's public key blob and `<ta_pub_key_blob.der>` is a filename to put the TA's public key blob received after a successful request. Upon receiving the request, the key exchange with the TA is performed in three phases.

5.5.2.1 Phase 1: Parameter Sanity

1. CA sends a 259-byte public-key blob, which includes 256 bytes of modulus and 3 bytes of exponent, along with a buffer to be populated with TA's public key.
2. Mismatch in parameters results in `TEE_ERROR_BAD_PARAMETERS` and the request is rejected.

5.5.2.2 Phase 2: Registry Lookup and Update

1. The TA looks up the caller CA's UUID in the client registry stored in the secure memory. If the Universally Unique Identifier (UUID) is present, no changes are made to the client registry.
2. If not found, the TA adds the tuple (UUID, modulus, exp) to the client registry. In case there is not enough memory to add a new entry to the registry, the request is rejected with `TEE_ERROR_OUT_OF_MEMORY`.

5.5.2.3 Phase 3: Key Provisioning

1. The TA generates or reconstructs the TA RSA key-pair in the secure memory.
2. Public half (256-byte modulus || 3-byte exponent) is copied into the buffer shared by CA.

3. CA copies the TA's public key to the file <ta_pub_key_blob.der>.

Successful Outcome. The caller receives the TA's public key and can now use it to wrap AES keys for future configuration packages. The TA adds the CA UUID to the client registry and logs the CA's public key for signature verification.

5.5.3 vFPGA Configuration

The vFPGA configuration sequence is kicked off when the CA running in the Linux VM calls

```
./ca_executable configure <enc_pkg.bin> <vfpga_id>
```

where <enc_pkg.bin> is a pre-built, cryptographically protected *client package* and <vfpga_id> (1 or 2) selects the target reconfigurable partition on the ZCU 102 Board.

Internally, based on the configure command, the CA, opens a GP session to the TA inside OP-TEE copies the package into a shared memory buffer, and supplies three parameters:

1. a pointer / size pair for the package blob, and
2. the requested vFPGA identifier.

From this point onward, every security-critical action happens in the secure world. The TA parses the package, authenticates the caller, and decrypts the bitstream. If all checks succeed, the TA hands the plaintext to the pTA that ultimately invokes the ARM Trusted Firmware, which calls the PMU to get the vFPGA configuration completed.

Before going into the details regarding configuration steps, it is worthwhile to present the contents of the Client Package. Figure 15 describes the package contents in their byte order. It includes four entities. Starting with a 60-byte header, the package further contains a 256-bit wrapped key used to encrypt the bitstream, a 256-bit signature to authenticate the CA, and finally the encrypted bitstream with variable length. Table 8 describes each of these entities in detail.

The 60-byte header of the client package holds 9 different fields that enable the configuration process, as shown in Figure 15. These fields and their utility are described in detail in Table 9.

Once the configuration request reaches the TA, it is done in six phases as described below.

5.5.3.1 Phase 1: Parameter and Resource Validation

This is the first phase of configuration after the TA receives the configuration request from the CA, along with the CA's package (as a TEEC memory reference) and the vFPGA-ID (as a TEEC value). The goal of this phase is to have a fully validated copy of the package in the secure memory with pointers to the wrapped key, Signature, and encrypted bitstream.

During this phase, the configuration can be rejected due to various reasons, including bad parameter types, unsupported vFPGA-ID, vFPGA being in an occupied state, invalid header, or impossible size fields resulting in TEE_ERROR_BAD_FORMAT or TEE_ERROR_BAD_PARAMETERS.

In this phase, the TA performs the following steps.

1. **TEE Parameter Validation.** The TA validates the TEE parameter types and rejects the request if the TEEC call from the client is malformed.
2. **vFPGA-ID verification.** Verify that the vFPGA passed is valid. In our setup, there are two reconfigurable partitions with valid values {1,2}.

Table 8: Description of different fields in the client package.

Field	Size (Bytes)	Position	Description
Header	60	[0:59]	The package header contains metadata for the package that self-describes the rest of the file. It (1) binds the package to a specific client (UUID), (2) enables the TA to perform sanity checks before allocating memory, and (3) supplies the nonce+tag needed for Galois counter mode (GCM) authentication. Figure 15 and Table 9 describe the content of the header in detail.
256-bit Wrapped Key	32	[60:91]	Rivest-Shamir-Adleman Public-Key Cryptography Standards (RSA-PKCS)#1-v1.5 encryption of the 32-byte Advanced Encryption Standard (AES) session key. Wrapping the AES key protects it in transit so that only the TA, which holds the matching private key, can recover it. Anyone without the TA's secret key learns nothing about the AES key, in turn protecting the bitstream.
256-bit Signature	32	[92:123]	The Signature RSA-PKCS#1-v1.5-Secure Hash Algorithm (SHA)-256 over the wrapped AES key. It is computed with the CA's private RSA key. It gives the TA cryptographic proof that the wrapped key in the package is coming from the client requesting the configuration. The signature eliminates the risk of a malicious entity substituting a different AES key.
Encrypted Bitstream	Variable	[124:]	Advanced Encryption Standard with Galois/Counter Mode 256-bits (AES-256-GCM) encryption of the raw partial bitstream. It is authenticated with the tag stored in the header, using the IV and the unwrapped AES key. The encryption provides both confidentiality (nobody can peek at the proprietary hardware design) and integrity (bit-flips or replay will be detected by the GCM tag). The IV in the header ensures every package is unique, even if two bitstreams happen to be identical.

3. **vFPGA Status Check.** Verify that the vFPGA requested is free. If occupied, the configuration is rejected. This prevents one client from clobbering another's design.
4. **Header Parsing.** Parse the 60-byte header and validate the entities like MAGIC or VERSION. Reject if anything is out of range or does not match the file length.
5. **Crypto Setup.** Locate the Wrapped key, Signature, and bitstream ciphertext.
6. **Secure Memory Port.** Copy the package from non-secure shared memory to a fresh secure buffer. This ensures that only trusted entities can access the bitstream.

5.5.3.2 Phase 2: Authentication and Authorization

Now, the validation of the package header is complete. In this phase, the TA needs to authenticate whether the CA is genuinely allowed to reconfigure vFPGA. At the start of this phase, the TA has the UUID of the CA, the 256-bit wrapped key, the 256-bit signature, and the TA's own private key.

Table 9: Description of different fields in the client package header.

Field	Size (Bytes)	Position	Description
MAGIC	4	[0:3]	The MAGIC bits are a known constant to let the TA decide if it should go ahead with parsing the header. This enables an *early reject* before other authentication and cryptography steps. If the value is wrong, the TA rejects the package with `TEE_ERROR_BAD_FORMAT` without reading the rest of the header.
VERSION	2	[4:5]	The VERSION value is for version maintenance of the setup. It can be bumped whenever the header layout or any cryptographic primitive changes. The TA can therefore keep backward compatibility by accepting old versions while adding code for new ones, or it can choose to hard-fail unknown versions.
H_LEN	2	[6:7]	The total length of the header, including this field itself. The packaging script first packs a dummy 0 here, appends the variable data, then patches the correct length, which is always 60 bytes with the current layout. On the TA side, this allows the parser to jump straight from the header to the wrapped key without hard-coding any numbers.
WK_LEN	2	[8:9]	Size (bytes) of the RSA-encrypted AES session key. For RSA-2048 with PKCS#1 v1.5 padding, this is always 256 bytes. Putting the length here future-proofs the format if there are subsequent changes to the cryptographic primitives.
SIG_LEN	2	[10:11]	Size (bytes) of the RSA signature that protects the wrapped key. In our setup, this value is 256, but explicitly carried to provide flexibility for subsequent changes.
C_LEN	4	[12:15]	Length of the AES-GCM ciphertext that encloses the partial bitstream from the client. A 4-byte field theoretically supports bitstreams of up to 4GiB in size. If all the checks for configuration are passed, the TA allocates exactly this many bytes in the secure memory for decryption and rejects the package if the remaining file size does not match.
UUID	16	[16:31]	The UUID included in the header is a 128-bit identifier of the Client Application that owns this package. During decryption, the TA looks this value up in the client registry. The UUID of the client requesting configuration should match the UUID in the package. If no match is found, the TA refuses to proceed (`TEE_ERROR_ACCESS_DENIED`).
IV	12	[32:43]	The nonce used for AES-256-GCM encryption of the bitstream.
TAG	16	[44:59]	The authentication tag output by AES-GCM. The TA copies it into a local buffer, passes it to `TEE_AEDecryptFinal()` GP API, and thereby verifies both integrity and authenticity of the bitstream in a single call. Any mismatch (e.g., flipped bit, replay attack, wrong key) fails the decryption, and the TA returns `TEE_ERROR_SECURITY`.

The goal of this phase is to unwrap a verified 256-bit AES-GCM encryption key in secure memory and ensure that it was signed by the CA registered under the matching UUID. In this phase, the configuration can be rejected due to several reasons, including the absence of the UUID from TA's client registry, RSA decryption error, RSA signature check failure resulting in `TEE_ERROR_SECURITY` or `TEE_ERROR_ACCESS_DENIED`.

The following are the steps performed by the TA in this phase.

1. **Client UUID Validation.** Compare the UUID of the requesting client with that from the package header. This ensures the caller previously exchanged public keys and enforces CA identity. If there is no match, the configuration request is rejected.
2. **Key Unwrapping.** RSA-decrypt the 256-byte wrapped key in the package using the TA's private key to obtain the AES-256 encryption key. If the unwrapping fails, the configuration request is rejected.
3. **Signature Verification.** RSA-verify the 256-byte Signature using the CA's public key. If the verification fails, the configuration request is rejected.

5.5.3.3 Phase 3: Decryption and Bitstream-Integrity Check

At the start of the third phase, the TA has an unwrapped 256-bit AES key along with the 12-byte IV and 16-byte tag from the package header. The goal of this phase is to ensure that the partial bitstream is untampered with a GCM check and to decrypt the encrypted bitstream.

The goal of this stage is to verify that the bitstream has not been tampered with and get a plaintext bitstream in the secure memory region.

In this phase, the configuration can be rejected due to several reasons, including a corrupted nonce, bit-flipped ciphertext, or memory issues while allocating the buffers, resulting in `TEE_ERROR_SECURITY` or `TEE_ERROR_OUT_OF_MEMORY`.

In this phase, the TA performs the following steps.

1. **Plaintext Buffer Allocation.** GP APIs restrict in-place decryption of the ciphertext. Thus, a buffer in the secure memory is created for the plaintext.
2. **Bitstream Decryption and Integrity Check.** AES-GCM (`TEE_AEDecryptFinal()` enables the integrity checks using the nonce and tag, while the decryption is performed with the unwrapped key. If the tag verification fails, configuration is aborted with the error `TEE_ERROR_SECURITY`

5.5.3.4 Phase 4: Hand-off to pTA for Configuration

The successful completion of the previous phase results in a decrypted bitstream in the secure memory. The goal of this phase is to initiate the bitstream reconfiguration. To this end, the TA hands off the configuration to pTA, which in turn invokes the ARM Trusted Firmware via the CROSSCON Hypervisor.

The configuration can be rejected at this phase due to several reasons, including pTA session failure, SMC routing error. In this phase, the pTA performs the following steps.

1. **Spin-lock Acquisition.** The TA acquires a spin-lock to ensure only one CA invokes a configuration session at a time.
2. **pTA Invocation.** Open an internal TEE session to pTA, requesting configuration. The pTA validates the parameters and rejects them if the parameters are malformed.

3. **SMC Invocation.** pTA invokes the Secure Monitoring Call (SMC) to the ARM Trusted Firmware executing in EL3 privilege level. From pTA, the CROSSCON Hypervisor catches this SMC call and forwards it transparently to the ARM Trusted Firmware.

5.5.3.5 Phase 5: Hand-off to the ARM Trusted Firmware for Configuration

In the previous phase, the ARM Trusted Firmware (TF-A) is invoked with the custom SMC call to configure the bitstream. The goal of this phase is to perform the final steps and get the bistream configured on the vFPGA.

Configuration failures in this phase can arise from SMC routing issues or the PMU's refusal to load (e.g., invalid bitstream). In this phase, the ARM Trusted Firmware and PMU perform the following steps.

1. **Sanity Checks.** The ARM Trusted Firmware receives the SMC from the pTA via the CROSSCON hypervisor and performs the following sanity checks:
   (a) If the parameters of the SMC are malformed, it rejects the request.
   (b) It checks the address passed as the bitstream location and proceeds only when it is valid.
2. **PMU Invocation.** Internally, the SMC handler in the TF-A invokes `pm_fpga_load()` call exposed by the PMU. This is the only entity that is allowed to talk to the FPGA configuration port after boot. It performs the partial reconfiguration of the selected reconfigurable partition. If the configuration is successful, it returns `TEE_SUCCESS`, provides an acknowledgment with an error message.
3. **Acknowledgment.** The acknowledgment from PMU goes back to all the cascading callers, the ARM Trusted Firmware, the CROSSCON Hypervisor, the pTA, and the TA.
4. **Spin-lock Release.** The TA releases the spin-lock.

5.5.3.6 Phase 6: Book-keeping and Clean-up

At the start of this phase, vFPGA is configured, or the request has failed, and the error code has been propagated to the TA. The goal of this step is to ensure that the TA reflects that state, propagates the acknowledgement to the CA, and frees resources.

The reasons for failure in this phase are unlikely, but can be due to errors in the memory-free operations. In this phase, the TA performs the following steps.

1. **vFPGA Status Update.** Upon successful configuration, the TA updates the status of the vFPGA-ID, marking it occupied by the CA's UUID.
2. **Memory Hygiene.** Free all transient objects, operations, and heap buffers. This ensures memory hygiene and prevents unintended leaks.
3. **Acknowledgment Forwarding.** Return `TEE_SUCCESS` or error code to the CA.

5.5.3.7 Successful Outcome

The partial bitstream from the CA is configured to the requested vFPGA, and the status is updated.

5.5.4 vFPGA Deallocation

The vFPGA deallocation sequence is kicked off when the CA running in the Linux VM calls

```
./ca_executable free <vfpga_id> <enc_pkg.bin>
```

where <vfpga_id> (1 or 2) selects the target reconfigurable partition on the ZCU 102 Board and <enc_pkg.bin> is a pre-built, cryptographically protected client package. The purpose of reusing the client package is to have an authentication that the client is allowed to free the specified vFPGA.

5.5.4.1 Phase 1: Parameter sanity

The parameter checks at the TA are the same as the ones done in the vFPGA configuration.

5.5.4.2 Phase 2: Ownership verification

In this phase, the TA extracts the UUID from the header and compares it against the vFPGA ID state maintained in the secure memory. A mismatch indicates the caller CA does not own the specified vFPGA and results in the request getting rejected with `TEE_ERROR_ACCESS_DENIED`.

5.5.4.3 Phase 3: State update

The TA frees the vFPGA by configuring a default bitstream loaded in the secure memory and zero-fills the 16-byte owner slot for the specified vFPGA, marking the partition *free*.

5.5.4.4 Successful Outcome

The specified vFPGA is free and available for reconfiguration.

5.6 Implementation

In this section, we describe the implementation details to realize the setup. First, we look at the approach used to build the CROSSCON Hypervisor, the TA, and the CAs. Second, we discuss the communication mechanism used between the CAs, TA, and pTA. Third, we look at the cryptographic primitives used by the TA. Finally, we present the partial bitstream configuration mechanism using a custom Secure Monitoring Call (SMC).

5.6.1 Build

In this section, we look at the build configurations of the key components, namely the CROSSCON Hypervisor, TA, pTA, and the CAs.

5.6.1.1 CROSSCON Hypervisor

The CROSSCON Hypervisor partitions the resources between the Linux VM and the OP-TEE VM. To this end, it uses a configuration file to set up the memory region. In this section, we briefly explain the configuration used in the implementation.

```
#include <config.h>

// Linux Image
VM_IMAGE(linux_image, "../lloader/linux-zcu.bin");
// Linux VM configuration
struct vm_config linux = {
    ...
    // details related to VM-loading
```

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

VM_IMAGE(optee_os_image, "../optee_os/optee-aarch64/core/tee-pager_v2.bin");
struct vm_config optee_os = {
    // details related to VM-loading

    .platform = {
        .cpu_num = 1,
        .region_num = 1,
        .regions = (struct mem_region[]) {
            {
            // Region 1: 16MB of secure memory region in the RAM
            .base = 0x60000000, .size = 0x01000000, .phys = 0x60000000,
                .place_phys = true,
            },
        },
        .ipc_num = 1,
        .ipcs = (struct ipc[]) {
            {
                .base = 0x42000000, .size = 0x00800000, .shmem_id = 0,
            }
        },
        ...
        // dev_num and other details
        }
    },
};

struct config config = {
    CONFIG_HEADER
    .shmemlist_size = 1,
    .shmemlist = (struct shmem[]) {
        [0] = { .size = 0x00800000, .phys = 0x42000000, .place_phys = true },
    },
```

```
    .vmlist_size = 1,
    .vmlist = {&optee_os}
};
```

Listing 5.1: CROSSCON Hypervisor memory configuration.

Listing 5.1 presents the relevant excerpts from the CROSSCON Hypervisor build. We can note that in this implementation, the shared memory region starts at 0x42000000 and is 8MiB in size. The CA loads the client package in these regions. Further, the secure memory region starts at 0x60000000 and is 16MiB large. The TA copies the package in this region and stores all the operations and data here. Note that the secure memory has `.place_phys=true` to ensure that the physical address of the bitstream passed by the pTA matches the location seen by the TF-A [29].

5.6.1.2 Trusted Application and pseudo TA

The TA is built with the following flags.

- `TA_FLAG_SINGLE_INSTANCE`: Only one global instance of the TA is ever loaded into secure memory, no matter how many clients open a session.
- `TA_FLAG_MULTI_SESSION`: Multiple clients can have sessions with the single instance of the TA.
- `TA_FLAG_INSTANCE_KEEP_ALIVE`: The TA instance is kept in secure memory even after the last session closes and destroyed only when the OS shuts down. This is helpful to maintain the client registry and vFPGA states.

The build also allows for a configurable size for the TA stack and heap through `TA_STACK_SIZE` and `TA_DATA_SIZE` parameters. The pTA is part of `optee_os/core` linked into the secure-world kernel with `CFG_PTA_FPGA_MANAGER` = y during build. Both the TA and pTA executables are signed by OP-TEE is at build time and verified by the secure loader during early boot.

5.6.1.3 Client Applications

The Client Applications are built as a buildroot package in the setup. The CAs have different invocations based on the kind of request from the TA: `CMD_GET_KEY`, `CMD_GET_STATUS`, `MY_TEST_CMD_CONFIGURE`, or `CMD_FREE`.

5.6.2 Communication

The communication between the CA and TA, and the TA and the pTA, uses the GP APIs end-to-end. In this section, we present a brief outline of the APIs used in the implementation.

CA-TA communication.

1) `TEEC_InitializeContext()`: discover TEE driver.
2) `TEEC_RegisterSharedMemory()`: obtain a buffer for the package.
3) `TEEC_OpenSession()` : SMC#0 entry to OP-TEE.
4) `TEEC_InvokeCommand()`: either `CMD_GET_KEY`, `CMD_GET_STATUS`, `MY_TEST_CMD_CONFIGURE`, or `CMD_FREE`.
5) `TEEC_CloseSession()`, `TEEC_FinalizeContext()`: clean-up.

TA-pTA communication.

1) `TEE_Malloc()`: duplicate package into secure memory, failure returns `TEE_ERROR_OUT_OF_MEMORY`.
2) `TEE_OpenTASession()`, `TEE_InvokeTACommand()`, `TEE_CloseTASession()`: hand plaintext to the pTA.

5.6.3 Cryptography

Cryptographic operations in the TA are implemented entirely with the GP Internal APIs that OP-TEE exposes in `tee_internal_api.h`.

Key hierarchy.

- **TA RSA-2048 keypair.** Either generated at first `get-key` via `TEE_GenerateKey()`, or reconstructed from cached key. The TA stores the modulus and public exponent in static globals in the secure memory (`ta_mod`, `ta_exp`) so it can echo them back to any CA.
- **CA RSA-2048 public key.** The public half (259-byte [modulus ||exponent blob]) is kept in the TA's dynamic registry in the secure memory.
- **Session AES-256 key.** Randomly generated by the CA on their end offline. The TA receives the wrapped version with its private key.

Algorithms and API calls. Table 10 describes the GP Internal API calls that invoked across various steps.

The error propagation also follows GP conventions:

- `TEE_ERROR_SECURITY` for any MAC/signature failure.
- `TEE_ERROR_BAD_FORMAT` for malformed input.
- `TEE_ERROR_OUT_OF_MEMORY` if the secure heap cannot satisfy a buffer or transient-object request.

5.6.4 SMC Handling

A Secure Monitor Call (SMC) is the ARM equivalent of a system call for the highest privilege level (EL3). Executing the assembler instruction `smc #imm` causes the CPU to trap into the *secure monitor*, passing up to 8 64-bit arguments in registers `x0–x7`. Registers `x0–x3` are mandatory, while `x4–x7` can be used based on requirement. According to the ARM SMC Calling Convention (SMCCC), the first register (`x0`) encodes whether the call is *32-bit or 64-bit*, and finally which service owner should handle it.

One particular owner value (`0x02`) reserved for vendor-defined functions anything that is specific to a given SoC but not covered by ARM-standard services. In our implementation, vFPGA configuration is such a service belonging to the Silicon-Partner owner class (SiP). Thus, for partial bitstream configuration, pTA encodes the call as `SMC_FID_FPGA_LOAD = 0xC3200200` which is known to TF-A. The call originates in the pTA (EL1), crosses the EL2 hypervisor unmodified, lands in the ARM Trusted Firmware SiP runtime service (EL3), and finally reaches the Platform-Management Unit firmware (PMU FW), which finally configures the partial bitstream.

1) SMC request from pTA. After re-checking the buffer alignment, the pTA fills an `smc_args` structure as described in the code excerpt in Listing 5.2. `cache_op()` flushes the secure DDR range so the

Table 10: GP Internal API calls used by the TA for cryptographic operations.

Purpose	GP API call(s)	Notes
Allocate key object	`TEE_AllocateTransientObject()`	Creates an empty key handle for either `TEE_TYPE_RSA_KEYPAIR` (TA & CA keys) or `TEE_TYPE_AES` (runtime session key).
Generate TA RSA-2048 key	`TEE_GenerateKey()`	Once-per-boot operation; skipped if a private key is cached.
Populate object with key bytes	`TEE_InitRefAttribute()` `TEE_PopulateTransientObject()`	Copies modulus/exponent (and private exponent for the TA key) from into the transient object.
Allocate algorithm context	`TEE_AllocateOperation()`	Reserves a crypto engine for `TEE_ALG_RSAES_PKCS1_V1_5`, `TEE_ALG_RSASSA_PKCS1_V1_5_SHA256`, `TEE_ALG_SHA256` or `TEE_ALG_AES_GCM`.
Bind key to operation	`TEE_SetOperationKey()`	Attaches the RSA or AES key object to the operation handle before use.
Unwrap AES session key	`TEE_AsymmetricDecrypt()`	`TEE_ALG_RSAES_PKCS1_V1_5`; converts the 256-byte wrapped key into a 32-byte secret.
Compute SHA-256 digest	`TEE_DigestDoFinal()`	Hardware-assisted SHA-256 over the wrapped key; feeds signature verify.
Verify RSA signature	`TEE_AsymmetricVerifyDigest()`	`TEE_ALG_RSASSA_PKCS1_V1_5_SHA256`; authenticates the wrapped key against the CA's public key.
Initialize AES-GCM session	`TEE_AEInit()`	Loads the 96-bit IV from the header and sets the tag length to 128 bits.
Decrypt & authenticate bitstream	`TEE_AEDecryptFinal()`	Processes the ciphertext and verifies the 16-byte tag in one call.

PMU sees the most recent data. The interface supports bitstreams up to 4GiB. The actual value of `cipher_len` is derived from the package header.

```
#define SMC_FID_FPGA_LOAD U(0xC3200200) /* SiP, fast, -64bit */

struct smc_args a = {
    .a0 = SMC_FID_FPGA_LOAD, /* function ID */
    .a1 = phys_addr, /* PA of bitstream */
    .a2 = cipher_len, /* byte length */
    .a3 = XILINX_ZYNQMP_PM_FPGA_PARTIAL /* flag: partial reconfig */
};

cache_op(phys_addr, cipher_len, DCACHE_FLUSH);

tee_smc_call(&a); /* trap to EL3 via SMC #0 */

return a.a0 ? TEE_ERROR_GENERIC : TEE_SUCCESS;
```

Listing 5.2: Code excerpt from the pTA issuing the SMC call.

2) SMC trap and forwarding by the CROSSCON Hypervisor. The SMC request from the pTA lands up in the Hypervisor. Upon checking the correctness of the SMC function ID, the handler in the hypervisor forwards the request transparently to TF-A, as shown in the code excerpt in Listing 5.3.

```
/* sdtz.c ------------------------------------------------------ */

static int handle_secure_smc(struct vcpu *vcpu){

    struct smc_res r;
    unsigned long x0 = vcpu_readreg(vcpu, 0);
    unsigned long x1 = vcpu_readreg(vcpu, 1);
    unsigned long x2 = vcpu_readreg(vcpu, 2);
    unsigned long x3 = vcpu_readreg(vcpu, 3);

    smc_call(x0, x1, x2, x3, &r); /* EL2 →EL3 */
    vcpu_writereg(vcpu, 0, r.x0); /* propagate result */

    return HYP_SMC_HANDLED;
}
```

Listing 5.3: Code excerpt from Hypervisor pass-through handler.

3) Custom SIP handling in TF-A. The ARM Trusted Firmware provides support for adding custom SiP services, registered via `DECLARE_RT_SVC`. We add a custom SiP handler that decodes the function ID and delegates to PMU FW as shown in the code excerpt in Listing 5.4. `pm_fpga_load()` is part of the Xilinx PMU firmware API, and the flag `XILINX_ZYNQMP_PM_FPGA_PARTIAL` selects *partial* (dynamic) reconfiguration mode.

```
/* custom_sip_svc.c ------------------------------------------- */

static uintptr_t fpga_load_handler(uint32_t smc_fid, uint64_t addr,
                                   uint64_t size, uint64_t flags,
                                   void *handle){

    enum pm_ret_status st = pm_fpga_load(addr, (uint32_t)size,
                                         (uint32_t)flags);

    return SMC_RET1(handle, st); /* 0 = OK, else PMU error */
}
```

Listing 5.4: Code excerpt from ARM Trusted Firmware custom SiP handler.

PMU FW sets its own status code (`PM_OK`, `PM_ERR_BUSY`, `PM_ERR_INVALID_PARAM`, etc.) which bubbles up unchanged through TF-A, the hypervisor, and the pTA back to the user TA. The mapping inside the pTA is:

- `PM_OK` (0): TA sends `TEE_SUCCESS` to the CA.
- `PM_ERR_BUSY`: TA sends `TEE_ERROR_BUSY` to the CA.
- `PM_ERR_INVALID_PARAM`: TA sends `TEE_ERROR_BAD_PARAMETERS` to the CA.
- other negative: TA sends `TEE_ERROR_GENERIC` to the CA.

5.7 Security Analysis

Table 11 provides a rundown of the security-violating actions that can be performed by CA. It also describes how our implementation thwarts these attempts.

Table 11: Security Analysis of the setup using different scenarios that can be attempted by a CA.

Scenario	Allowed?	Notes
Steal and pass the package of another CA for vFPGA configuration	✗	TA compares header-UUID with caller CA's UUID; `TEE_ERROR_ACCESS_DENIED`.
Invoke configuration without registering.	✗	UUID not present in registry, early abort.
Tamper a bit in the MAGIC, VERSION, or size fields.	✗	Header sanity check; `TEE_ERROR_BAD_FORMAT`.
Replay a previously observed package to load a stale bitstream.	✗	GCM tag fails if the wrapped AES key differs. If key and tag are unchanged, the FPGA already carries that design, i.e., no additional effect.
Modify one byte in the encrypted bitstream.	✗	GCM tag mismatch;`TEE_ERROR_SECURITY`, plain text is never forwarded.
Replace wrapped key and recompute ciphertext without CA's private key.	✗	RSA-verify fails at TA, configuration rejected without decryption.
Free a vFPGA that belongs to another CA.	✗	Free handler checks owner table; `TEE_ERROR_ACCESS_DENIED`.
Read plaintext bitstream.	✗	Plaintext resides only in secure memory, inaccessible to CAs.
Modify the buffer while the TA decrypts it.	✗	TA copies the package into a private, secure buffer before use; `TEE_ERROR_ACCESS_DENIED`.
Pass a bitstream to overflow secure memory	✗	Request rejected at header check.
Invoke two `configure` calls in parallel.	✗	Spin-lock during configuration ensures sequentiality.
Alter registers during the SMC request of another CA.	✗	Spin-lock prevents cross-session contamination.

5.8 Demonstrator

Our demonstrator is implemented on top of the AMD Xilinx ZCU102 Evaluation Kit, deploying a Zynq UltraScale+ MPSoC with a quad-core ARM Cortex-A53 processor (referred to as Application Processing Unit or APU), a dual-core Cortex-R5F real-time processor (referred to as Real-time Processing Unit or RPU), and an FPGA fabric by AMD. The platform supports ARM TrustZone technology. Akin to the architecture presented in Figure 14, the FPGA is partitioned into three fixed-size regions: a static PL and two vFPGAs that can be allocated to clients. The static PL implements the clock management for the vFPGAs using Xilinx IP cores. Each vFPGA has a dedicated I/O interface such that vFPGAs can be controlled by different masters. In addition to these, the demonstrator has the following components.

- Two client applications, CA1 and CA2, with executables in the Linux VM. CA1 has a client package b1.pkg while CA2 has a client package b2.pkg. The bitstreams b1 and b2 in the packages have identifiable physical properties that can be verified on successful configuration (in addition to the PMU acknowledgements).

Table 12: The ordered steps to be performed in the Demonstration, along with the expected Results.

#	Demonstrator step (in order)	Result
1.	CA1/CA2 runs the configure command with b1.pkg and vFPGA-ID 1	Configuration rejected, TA complains unknown CA.
2.	CA1/CA2 perform key exchange	TEE_SUCCESS
3.	CA1/CA2 run status commad	Satus printed on terminal
4.	CA1 runs the configure command with b1.pkg and vFPGA-ID 1	TEE_SUCCESS
5.	CA2 runs the configure command with b2.pkg and vFPGA-ID 1	Configuration rejected, TA complains that the vFPGA is not free.
6.	CA2 runs the configure command with b1.pkg and vFPGA-ID 2	Configuration rejected, TA b1.pkg is invalid for CA2.
7.	CA2 runs the configure command with b2.pkg and vFPGA-ID 2	TEE_SUCCESS
8.	CA2 runs status command	Both vFPGAs occupied.
9.	CA2 runs the free command with b2.pkg and vFPGA-ID 1	Free operation rejected, TA complains that CA2 does not own this vFPGA.
10.	CA2 runs the free command with b2.pkg and vFPGA-ID 2	TEE_SUCCESS

- A trusted application TA and a pseudo TA on the OPTEE VM.
- CROSSCON hypervisor running on the ZCU 102 board, handling the two VMs.

Table 12 lists the ordered sequence of steps to be shown in the demonstrator and the expected results.

6 Conclusion

This document presents the final results of WP4 and it's contributions toward MS8 and MS9. The efforts focused on improving the security of IoT devices at the hardware level through the co-design of trusted hardware and software components integrated into the CROSSCON Stack. WP4 produced five innovations that, together, contribute to a more secure and adaptable IoT ecosystem.

First, several unified interfaces have been developed to simplify interaction with hardware primitives and provide consistent access to trusted services across heterogeneous platforms. Second, the CROSSCON SoC introduces a custom RISC-V–based architecture with strong isolation properties and support for TEE-like environments. Third, the Perimeter Guard mechanism enables secure communication over SoC interconnects and supports the sharing of hardware accelerators across isolated domains without compromising isolation. Fourth, the secure provisioning of FPGA-based accelerators has been optimized to ensure domain isolation within reconfigurable hardware fabrics. Finally, TAGShield has been developed to provide a persistent and deterministic defense against stack-based spatial memory errors by leveraging ARM's Memory Tagging Extensions (MTE).

These five components complement the CROSSCON Stack and can be used to develop more secure IoT systems, while emphasizing interoperability, modularity, and robustness. Using the solutions across the hardware-software stack allows enforcement of essential security guarantees, such as caller isolation, while preserving flexibility and performance. These innovations lay the groundwork for the secure deployment of IoT applications in environments with demanding security requirements.

References

[1] Arm Ltd. *Arm cryptocell-300 Machine Learning Processor*. URL: https://www.arm.com/products/silicon-ip-security/crypto-cell-300 (visited on 07/02/2025).

[2] Nordic Semiconducter. URL: https://docs.nordicsemi.com/bundle/nrf5_SDK_v17.1.1/page/group_cryptocell_api.html (visited on 07/02/2025).

[3] NXP. *RT600 Hash Engine*. July 2020. URL: https://www.nxp.com/docs/en/application-note/AN12834.pdf (visited on 07/02/2025).

[4] NXP. *AN12445 - asymmetric cryptographic accelerator Casper*. Sept. 2023. URL: https://www.nxp.com/docs/en/application-note/AN12445.pdf (visited on 07/02/2025).

[5] Renesas Electronics. *Renesas Security Engine Operational Modes*. Nov. 2024. URL: https://www.renesas.com/en/document/apn/renesas-security-engine-operational-modes (visited on 07/02/2025).

[6] Freescale Semiconductor. *ColdFire/ColdFire+ CAU and Kinetis mmCAU Software Library*. URL: https://www.nxp.com/docs/en/user-guide/CAUAPIUG.pdf (visited on 07/02/2025).

[7] Infineon. *PSoC 6 Peripheral Driver Library: Crypto (Cryptography)*. URL: https://infineon.github.io/psoc6pdl/pdl_api_reference_manual/html/group__group__crypto.html (visited on 07/02/2025).

[8] Infineon. *MTB CAT1 Peripheral Driver Library: Cryptolite (Cryptography)*. URL: https://infineon.github.io/mtb-pdl-cat1/pdl_api_reference_manual/html/group__group__cryptolite.html (visited on 07/02/2025).

[9] Biagio Peccerillo, Mirco Mannino, Andrea Mondelli, and Sandro Bartolini. ``A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives''. In: *Journal of Systems Architecture* 129 (2022), p. 102561.

[10] Lehlogonolo PI Ledwaba, Gerhard P Hancke, Hein S Venter, and Sherrin J Isaac. ``Performance costs of software cryptography in securing new-generation Internet of energy endpoint devices''. In: *IEEE Access* 6 (2018), pp. 9303–9323.

[11] ARM. *PSA Certified Crypto API 1.1*. Mar. 2023. URL: https://arm-software.github.io/psa-api/crypto/1.1/ (visited on 07/02/2025).

[12] Global Platform. *Global Platform Tee Internal Core API Specification V1.1.2.50*. June 2018. URL: https://globalplatform.org/wp-content/uploads/2018/06/GPD_TEE_Internal_Core_API_Specification_v1.1.2.50_PublicReview.pdf (visited on 07/02/2025).

[13] Ravikanth Pappu, Ben Recht, Jason Taylor, and Neil Gershenfeld. ``Physical one-way functions''. In: *Science* 297.5589 (2002), pp. 2026–2030.

[14] Xilinx. *EMBEDDEDSW*. Apr. 2024. URL: https://github.com/Xilinx/embeddedsw/blob/master/lib/sw_services/xilskey/src/xilskey_eps_zynqmp_puf.c (visited on 07/02/2025).

[15] Maxim Intergated. *MAX32520 chipdna secure arm cortex M4 microcontroller*. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/MAX32520.pdf (visited on 07/02/2025).

[16] PUFsecurrity. *Root of trust: Pufrt: PUF-based security IP Solutions*. Feb. 2025. URL: https://www.pufsecurity.com/products/pufrt/ (visited on 07/02/2025).

[17] PUFsecurity. *Crypto coprocessor: PUFCC: PUF-based security IP Solutions*. Feb. 2025. URL: https://www.pufsecurity.com/products/pufcc/ (visited on 07/02/2025).

[18] Synopsis. *Synopsys PUF Base/premium - Software*. URL: https://www.synopsys.com/dw/ipdir.php?ds=security-puf-ip-software (visited on 07/02/2025).

[19] Mohammad Ebrahimabadi, Mohamed Younis, and Naghmeh Karimi. ``A PUF-Based Modeling-Attack Resilient Authentication Protocol for IoT Devices''. In: *IEEE Internet of Things Journal* PP (July 2021), pp. 2327–4662. DOI: 10.1109/JIOT.2021.3098496.

[20] Prosanta Gope, Jemin Lee, and Tony Q. S. Quek. ``Lightweight and Practical Anonymous Authentication Protocol for RFID Systems Using Physically Unclonable Functions''. In: *IEEE Transactions on Information Forensics and Security* 13.11 (2018), pp. 2831–2843. DOI: 10.1109/TIFS.2018.2832849.

[21] Global Platform. *Global Platform Tee Internal Core API Specification V1.3.1*. July 2021. URL: https://globalplatform.org/wp-content/uploads/2021/03/GPD_TEE_Internal_Core_API_Specification_v1.3.1_PublicRelease_CC.pdf (visited on 07/23/2025).

[22] Ahmet Turan Erozan, Michael Hefenbrock, Michael Beigl, Jasmin Aghassi-Hagmann, and Mehdi B. Tahoori. *Image PUF: A Physical Unclonable Function for Printed Electronics based on Optical Variation of Printed Inks*. Cryptology ePrint Archive, Paper 2019/1419. 2019. URL: https://eprint.iacr.org/2019/1419 (visited on 07/23/2025).

[23] ``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.

[24] Francesco Gringoli, Matthias Schulz, Jakob Link, and Matthias Hollick. ``Free Your CSI: A Channel State Information Extraction Platform For Modern Wi-Fi Chipsets''. In: *Proceedings of the 13th International Workshop on Wireless Network Testbeds, Experimental Evaluation Characterization*. WiNTECH '19. 2019, pp. 21–28. URL: https://doi.org/10.1145/3349623.3355477.

[25] Matthias Schulz, Daniel Wegemer, and Matthias Hollick. *Nexmon: The C-based Firmware Patching Framework*. 2017. URL: https://nexmon.org.

[26] 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.

[27] Raphael Spreitzer and Thomas Plos. ``On the applicability of time-driven cache attacks on mobile devices''. In: *International Conference on Network and System Security*. Springer. 2013, pp. 656–662.

[28] Cristiano Rodrigues, Daniel Oliveira, and Sandro Pinto. ``BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect''. In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024, pp. 3679–3696. DOI: 10.1109/SP54263.2024.00062.

[29] Marco Roveri, Emanuele Beozzo, Michele Grisafi, Alberto Tacchella, Žiga Putrle, João Sousa, David Cerdeira, and Nikhilesh Singh. *D2.3: CROSSCON Open Specification - Final*. 2025.

[30] *Beyond Semiconductor*. URL: https://www.beyondsemi.com/ (visited on 07/10/2025).

[31] *BA51 RISC-V core*. URL: https://www.cast-inc.com/processors/risc-v/ba51 (visited on 07/10/2025).

[32] *Spike RISC-V ISA Simulator,* URL: https://github.com/riscv-software-src/riscv-isa-sim (visited on 07/10/2025).

[33] Beyond Semiconductor. *Extended Spike RISC-V ISA Simulator*. URL: https://github.com/crosscon/riscv-isa-sim (visited on 07/10/2025).

[34] *RISC-V SPMP specification*. URL: https://github.com/riscv/riscv-spmp (visited on 07/10/2025).

[35] Sandro Pinto, José Martins, Manuel Rodríguez, Tilen Nedanovski, Ziga Putrle, and Matjaz Breskvar. ``Securing Embedded and IoT Systems with SPMP-based Virtualization''. In: *RISC-V Summit Europe 2024* (2024). URL: https://riscv-europe.org/summit/2024/media/proceedings/posters/177_poster.pdf (visited on 07/10/2025).

[36] YosysHQ. *Symbiyosys*. https://github.com/YosysHQ/sby. 2025.

[37] Marco Roveri nd Alberto Tacchella, Jure Mihelič, and Žiga Putrle. *D2.4: CROSSCON Formal Framework - Final*. 2025.

[38] Morris J Dworkin, Elaine Barker, James R Nechvatal, James Foti, Lawrence E Bassham, E Roback, James F Dray Jr, et al. ``Advanced encryption standard (AES)''. In: (2001).

[39] Morris J Dworkin. ``Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC''. In: (2007).

[40] Beyond Semiconductor. *CROSSCON SoC*. https://github.com/crosscon/crosscon_soc. 2025.

[41] Beyond Semiconductor. *CROSSCON SoC with CROSSCON Hypervisor and AES-GCM accelerator example*. https://github.com/crosscon/crosscon_soc/tree/main/examples/cs_hypervisor_with_aes_gcm_example. 2025.

[42] ARM. *Learn the Architecture - Providing Protection for Complex Software*. https://developer.arm.com/documentation/102433/0100. 2022.

[43] Andreas Hager-Clukas and Konrad Hohentanner. ``DMTI: Accelerating Memory Error Detection in Precompiled C/C++ Binaries with ARM Memory Tagging Extension''. In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. 2024, pp. 1173–1185.

[44] Xingman Chen, Yinghao Shi, Zheyu Jiang, Yuan Li, Ruoyu Wang, Haixin Duan, Haoyu Wang, and Chao Zhang. ``MTSan: A Feasible and Practical Memory Sanitizer for Fuzzing COTS Binaries''. In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, pp. 841–858.

[45] Floris Gorter, Taddeus Kroes, Herbert Bos, and Cristiano Giuffrida. ``Sticky Tags: Efficient and Deterministic Spatial Memory Error Mitigation using Persistent Memory Tags''. In: *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society. 2024, pp. 217–217.

[46] Jiwon Seo, Junseung You, Donghyun Kwon, Yeongpil Cho, and Yunheung Paek. ``ZOMETAG: Zone-based memory tagging for fast, deterministic detection of spatial memory violations on ARM''. In: *IEEE Transactions on Information Forensics and Security* (2023).

[47] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. ``SoftBound: Highly compatible and complete spatial memory safety for C''. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2009, pp. 245–258.

[48] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. ``Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors.'' In: *USENIX Security Symposium*. Vol. 10. 2009, p. 96.

[49] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. ``Stack Bounds Protection with Low Fat Pointers.'' In: *NDSS*. Vol. 17. 2017, pp. 1–15.

[50] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. ``Delta pointers: Buffer overflow checks without the checks''. In: *Proceedings of the Thirteenth EuroSys Conference*. 2018, pp. 1–14.

[51] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. ``Memory Tagging and how it improves C/C++ memory safety''. In: *arXiv preprint arXiv:1802.09517* (2018).

[52] LLVM Project. *LLVM 18.1.8 Release Notes*. URL: https://releases.llvm.org/18.1.8/docs/ReleaseNotes.html.

[53] Dirk Koch. *Partial Reconfiguration on FPGAs - Architectures, Tools and Applications*. Vol. 153. Lecture Notes in Electrical Engineering. Springer, 2013. ISBN: 978-1-4614-1224-3. DOI: 10.1007/978-1-4614-1225-0. URL: https://doi.org/10.1007/978-1-4614-1225-0.

[54] Christian Plessl and Marco Platzner. ``Virtualization of Hardware - Introduction and Survey''. In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA'04, June 21-24, 2004, Las Vegas, Nevada, USA*. Ed. by Toomas P. Plaks. CSREA Press, 2004, pp. 63–69.

[55] Stuart Byma, J. Gregory Steffan, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. ``FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack''. In: *22nd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2014, Boston, MA, USA, May 11-13, 2014*. IEEE Computer Society, 2014, pp. 109–116. DOI: 10.1109/FCCM.2014.42. URL: https://doi.org/10.1109/FCCM.2014.42.

[56] Suhaib A. Fahmy, Kizheppatt Vipin, and Shanker Shreejith. ``Virtualized FPGA Accelerators for Efficient Cloud Computing''. In: *7th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2015, Vancouver, BC, Canada, November 30 - December 3, 2015*. IEEE Computer Society, 2015, pp. 430–435. DOI: 10.1109/CLOUDCOM.2015.60. URL: https://doi.org/10.1109/CloudCom.2015.60.

[57] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. ``Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS''. In: *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. Ed. by Andrea C. Arpaci-Dusseau and Geoff Voelker. USENIX Association, 2018, pp. 107–127. URL: https://www.usenix.org/conference/osdi18/presentation/khawaja.

[58] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. ``A Survey on FPGA Virtualization''. In: *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*. IEEE Computer Society, 2018, pp. 131–138. DOI: 10.1109/FPL.2018.00031. URL: https://doi.org/10.1109/FPL.2018.00031.

[59] Furkan Turan and Ingrid Verbauwhede. ``Trust in FPGA-accelerated Cloud Computing''. In: *ACM Comput. Surv.* 53.6 (2021), 128:1–128:28. DOI: 10.1145/3419100. URL: https://doi.org/10.1145/3419100.

[60] Ghada Dessouky, Ahmad-Reza Sadeghi, and Shaza Zeitouni. ``SoK: Secure FPGA Multi-Tenancy in the Cloud: Challenges and Opportunities''. In: *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. IEEE, 2021, pp. 487–506. DOI: 10.1109/EUROSP51992.2021.00040. URL: https://doi.org/10.1109/EuroSP51992.2021.00040.

[61] Ilias Giechaskiel, Kasper Bonne Rasmussen, and Jakub Szefer. ``C3APSULe: Cross-FPGA Covert-Channel Attacks through Power Supply Unit Leakage''. In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1728–1741. DOI: 10.1109/SP40000.2020.00070. URL: https://doi.org/10.1109/SP40000.2020.00070.

[62] Ilias Giechaskiel, Kasper Bonne Rasmussen, and Ken Eguro. ``Leaky Wires: Information Leakage and Covert Communication Between FPGA Long Wires''. In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*. Ed. by Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim. ACM, 2018, pp. 15–27. DOI: 10.1145/3196494.3196518. URL: https://doi.org/10.1145/3196494.3196518.

[63] Dina Mahmoud and Mirjana Stojilovic. ``Timing Violation Induced Faults in Multi-Tenant FPGAs''. In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*. Ed. by Jürgen Teich and Franco Fummi. IEEE, 2019, pp. 1745–1750. DOI: 10.23919/DATE.2019.8715263. URL: https://doi.org/10.23919/DATE.2019.8715263.

[64] Falk Schellenberg, Dennis R. E. Gnad, Amir Moradi, and Mehdi B. Tahoori. ``An Inside Job: Remote Power Analysis Attacks on FPGAs''. In: *IEEE Des. Test* 38.3 (2021), pp. 58–66. DOI: 10.1109/MDAT.2021.3063306. URL: https://doi.org/10.1109/MDAT.2021.3063306.

[65] Mark Zhao and G. Edward Suh. ``FPGA-Based Remote Power Side-Channel Attacks''. In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 229–244. DOI: 10.1109/SP.2018.00049. URL: https://doi.org/10.1109/SP.2018.00049.