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

D4.1 CROSSCON Extensions to Domain Specific Hardware Architectures Documentation - Draft

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

Related WP	WP4	**Document Reference**	D4.1
Related Deliverable(s)	D4.2	**Dissemination Level (*)**	PU
Lead Participant	BEYOND, TUD	**Lead Author**	Žiga Putrle, Shaza Zeitouni
Contributors	BEYOND, UWU	**Reviewers**	Sandro Pinto, Luís Cunha, (UMINHO)
			Tilen Nedanovski, (BEYOND)

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

Document Information

List of Contributors	
Name	Partner
Žiga Putrle	BEYOND
Shaza Zeitouni	TUD
Richard Mitev	TUD
Marco Chilese	TUD
Lukas Petzi	UWU

Document History			
Version	Date	Change editors	Changes
0.1	11/03/2024	Shaza Zeitouni, Richard Mitev, Marco Chilese, Lukas Petzi, Žiga Putrle	Initial contributions
0.2	03/04/2024	Shaza Zeitouni, Richard Mitev, Marco Chilese, Lukas Petzi, Žiga Putrle	Refining content
0.3	24/04/2024	Shaza Zeitouni, Richard Mitev, Marco Chilese, Lukas Petzi, Žiga Putrle	Addressing review comments and making final adjustments
0.9	25/04/2024	Juan Alonso (ATOS)	Quality Assessment
1.0	30/04/2024	Hristo Koshutanski (ATOS)	Final version submitted

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable leader	Žiga Putrle (BEYOND), Shaza Zeitouni (TUD)	24/04/2024
Quality manager	Juan Alonso (ATOS)	25/04/2024
Project Coordinator	Hristo Koshutanski (ATOS)	30/04/2024

Table of Contents

List of Tables

List of Figures

List of Acronyms

Abbreviation / acronym	Description
ACC	Accelerator
AHB	Advance High-performance Bus
APB	Advance Peripheral Bus
API	Application Programming Interface
APLIC	Advanced Platform-Level Interrupt Controller
APU	Application Processing Unit
BA5x	Beyond Semiconductor’s line of RISC-V cores
D4.1	Deliverable number 1 belonging to WP 4.
D4.2	Deliverable number 2 belonging to WP 4.
DMA	Direct Memory Access
DQMEM	Data QMEM
DSP	Digital Signal Processing
EC	European Commission
FPGA	Field-Programmable Gate Array
HW	Hardware
IoT	Internet of Things
IQMEM	Instruction QMEM
MLIC	Multi-Layer InterConnect
MPU	Mmicroprocessor Unit
OTP	One Time Programmable
PG	Perimeter guard
PUF	Physically Unclonable Function
QMEM	Quick Memory
RAM	Random Access Memory
ROM	Read Only Memory
RoT	Root of Trust
RPU	Real-time Processing Unit
SoC	System on Chip
SPMP	S-mode Physical Memory Protection
SW	Software
TEE	Trusted Execution Environment
UART	Universal Asynchronous Receiver-Transmitter
vFPGA	Virtual FPGA
VM	Virtual Machine
vSPMP	Virtual SPMP
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 initial results of our work - including initial analysis, design and plan of future work. Our work can be divided into five related areas: (1) a custom RISC-V based SoC, called CROSSCON SoC, that can be used together with the CROSSCON Stack to provide strong security guarantees; (2) a unified interface that allows trusted services to access domain specific hardware, such us cryptographic accelerators or physical unclonable functions; (3) a mechanism that allows sharing of hardware (HW) accelerators between isolated domains without compromising the isolation between the domains; (4) a control-flow attestation accelerator that allows efficient pre-processing of execution traces; and, (5) a TEE-like environment on field-programmable gate arrays (FPGAs) that allows users to extend their trusted applications with trusted FPGA-accelerated tasks. By leveraging any of the aforementioned solutions together with the CROSSCON stack, one can improve the security of their IoT device.

This deliverable D4.1 contributes to the accomplishment of milestone MS4 “First version of the CROSSCON Stack components and extension primitives, and testbed”. As part of our future work, we plan to refine and build upon the solutions described in this document to provide proof of concept implementations, improve their maturity and obtain stronger security guarantees.

1 Introduction

1.1 Purpose of the document

The document outlines the objectives and methodologies of WP4, which encompasses various aspects crucial for enhancing the security and flexibility of IoT devices. Firstly, it emphasises the necessity of designing custom SoCs to attain greater control, transparency, and security, contrasting off-the-shelf SoCs with their limitations. Additionally, it discusses the importance of developing a common interface within the CROSSCON Stack for cryptographic primitives aimed at simplifying secure IoT device development across different platforms. Furthermore, it explores the challenges of sharing hardware modules, such as cryptographic accelerators, among isolated domains while maintaining strict isolation, proposing solutions like the Perimeter guard module. Lastly, it addresses the optimisation of FPGAs to accommodate multiple hardware accelerators while ensuring domain isolation, outlining strategies to secure FPGA configuration.

1.2 Relation to other project work

The work described in this document is closely related to work done as part of WP2, WP3 and WP5 as it contains the necessary insight for future work. Particularly, it is relevant to WP2 and WP3 for initial design and feasibility testing and to WP5 for integration into a cohesive platform. The research and development work in WP4 aligns with the overarching project objectives, emphasising the significance of enhancing hardware-level security within the CROSSCON Stack.

This deliverable D4.1 contributes to the accomplishment of milestone MS4 "First version of the CROSSCON stack components and extension primitives, and testbed", and it's closely related to the deliverable D4.2 "CROSSCON Extension Primitives to Domain Specific Hardware Architectures Initial Version", as it contains the implementation results described in this document.

1.3 Structure of the document

Next, we give an overview of the goals of WP4, which aim to enhance IoT device security by co-designing new HW/SW features, particularly focusing on domain-specific hardware and architecture. Following this, section 3 delves into extension primitives for trusted services such as cryptographic accelerators and APIs, highlighting research on hardware accelerators prevalent in IoT devices and analysing proposals for unified API sets. Section 4 extends on a novel machine-learning based control-flow attestation (CFA) scheme for IoT devices and describes the functionality of a CFA accelerator for efficient CFA attestation reports. Section 5 discusses the CROSSCON SoC, detailing its architecture, components, and the capability to add additional hardware modules. Moving on to section 6, the Perimeter Guard module is introduced, explaining its purpose and behaviour in sharing hardware modules/devices across security domains while maintaining isolation. In section 7, we explore extending trusted execution environment to FPGAs, focusing on key enablers such as advancements in FPGA virtualization, IP protection strategies, and the importance of bitstream verification for ensuring security in FPGA-based systems across diverse computing environments. We conclude in section 8 and provide our plan for future work.

2 Overview

The goal of WP4 is to improve 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. Thus far, our work can be divided into five related areas that can be put into a context of a HW-SW stack as shown in Figure 1. We discuss each area in detail in the following sections.

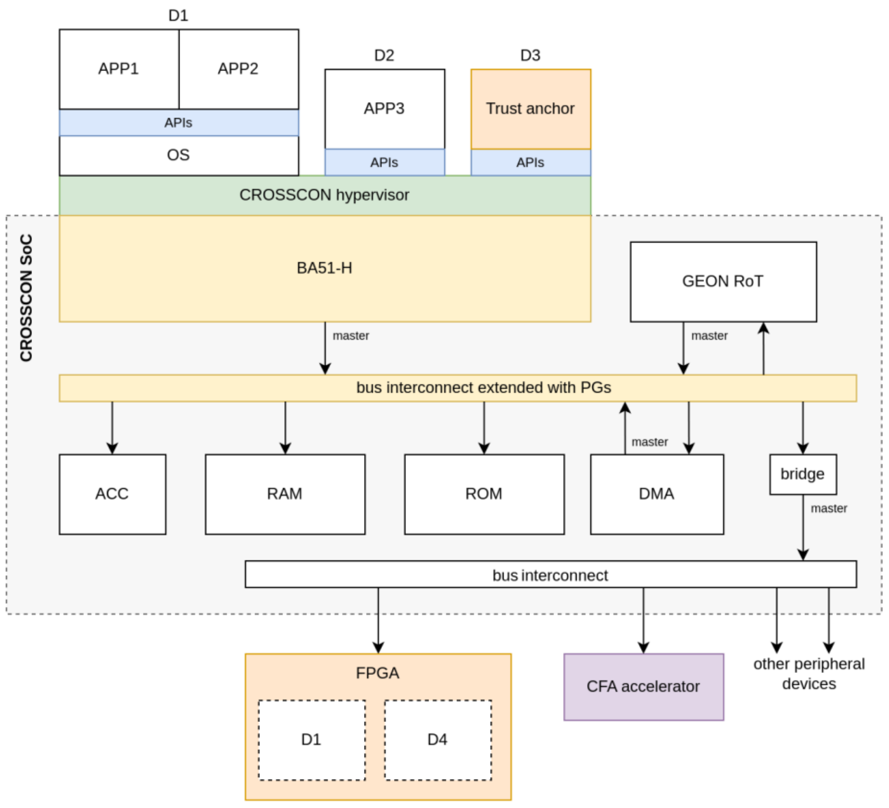

Figure 1: A simplified HW-SW stack

Figure 1 shows a simplified HW-SW stack that demonstrates how the main five areas of our work are related: (1) CROSSCON SoC (marked with grey), (2) a unified interface that allows Trusted services to access domain specific hardware (marked with blue), (3) set of HW features that enable sharing of HW accelerators without compromising isolation between domains (marked with yellow), (4) Control-Flow Attestation (CFA) accelerator that allows efficient pre-processing of execution traces (marked with purple), and (5) a Trust anchor that can be used to provide TEE-like environment on FPGA (marked with orange).

CROSSCON SoC (1): 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 the WP4, we are designing a custom SoC, called CROSSCON SoC, that can be used together with the CROSSCON Stack to improve security of IoT devices. The CROSSCON SoC includes 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).

A unified interface for trusted services (2): 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. As part of the WP4, we have investigated different vendor specific application programming interfaces (APIs) and libraries used across the industry. While numerous vendors deploy their own proprietary APIs for domain specific accelerators, there has been some efforts from both industry and academia to specify a unified API. These efforts have led to various proposals tailored to different devices classes and use cases. We investigate the two most popular APIs: the PSA Crypto API and Global Platform Cryptographic Operations API.

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 single domain (the safe way) or it is shared between domains, which can compromise the existing isolation (the unsafe way). As part of the WP4, we are looking into solutions that are easy to apply and would allow to share a module across domains while preserving the isolation guarantees. In chapter 6, we present a HW module, called Perimeter guard (PG), that seems as a promising solution that can address this problem.

CFA acceleration (4): CFA is a method employed for remotely assessing the validity of a program's execution. It encompasses the creation of a log detailing the execution traces of a program at runtime, which is then utilized by a verifier to attest the correct execution of the program and to create a report to prove the validity to a third party. To perform this efficiently, we are looking into possible HW accelerated solutions that would allow us to pre-process the execution log / traces for a given validation process.

TEE-like environment on FPGAs (5): 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. In WP4, our focus revolves around devising strategies to optimize FPGA utilization while ensuring strict isolation between different domains that share the FPGA fabric simultaneously. Section 7 delves into potential solutions addressing this challenge.

All five areas of our work address different problems that we encounter when designing an IoT device; but the suggested solutions complement each other, so we can use them as a part of a single solution, as demonstrated by the HW-SW stack in Figure 1. For example, we can use PGs in CROSSCON SoC to share HW accelerators across different virtual machines (VMs) while preserving isolation provided by the BA51-H core and CROSSCON Hypervisor. Furthermore, we can 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, we could setup control-flow attestation service that would use the CFA accelerator to attest that a program in TEE was executed as expected. And finally, 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.

An example of a security guarantee that we would like to support with our work is caller isolation. A *caller isolation* is a property of an interface (e.g. a library) that allows the interface to be used by several users (e.g. applications or VMs) without any interference between them. This is a great property for an interface to have, especially in a situation where the implementation of the interface uses a common resource (e.g. cryptographic HW accelerator or an FPGA). Note that because the functionality provided by the interface is usually implemented in software and hardware, we need to enforce caller isolation through the entire HW-SW stack. For example, Figure 10 shows how a call to a library could trigger a flow of information through an entire HW-SW stack. All five areas of our work can be used in some way to address this.

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

3 Extension Primitives for Trusted Services

To establish a cohesive set of extension primitives for Trusted Services, it's imperative to begin with a comprehensive overview of existing components. This includes cryptographic accelerators, domain-specific hardware, and APIs designed to streamline the diverse array of vendor-specific hardware interfaces. Consequently, our initial step involved researching the current landscape of hardware accelerators prevalent in IoT devices. We then scrutinized two prominent proposals aimed at creating a unified API set for heterogeneous IoT devices. Our objective was to gain insights into how these proposals could be leveraged and expanded to accommodate the innovative Trusted Services being developed within CROSSCON. This strategic approach not only facilitates interoperability within the CROSSCON Stack but also ensures compatibility across a broad spectrum of IoT devices.

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 CrpytoCell 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][4]. 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[5] 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)[6] 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[7] 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 a 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[8] or CryptoLite[9] 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, 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 hardware abstractions 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 requisite 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 applications 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 [10][11]. 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.

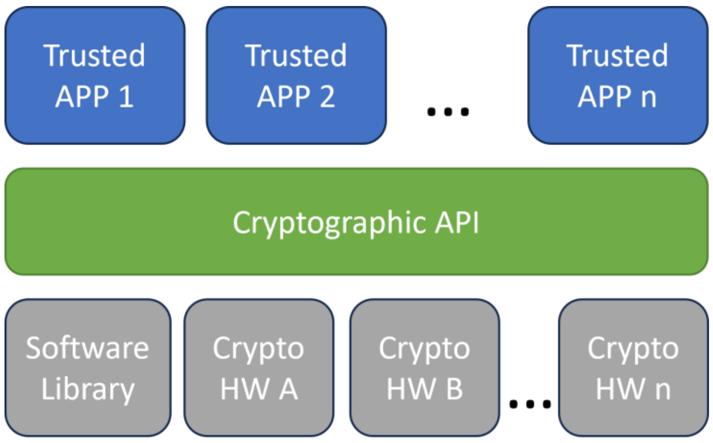

Figure 2:Unified Cryptographic Operations API

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 GlobalPlatform Cryptographic Operations API, with detailed findings presented subsequently.

3.2.1.1 PSA Crypto API

Arm's PSA Crypto API[12] 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.

The PSA API defines an interface accessible to applications, with mbedTLS serving as its software backend implementation. To accommodate domain-specific accelerators, such as hash accelerators or dedicated cryptographic co-processors, 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 D3.2, 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 GlobalPlatform Cryptographic Operations API.

3.2.1.2 GlobalPlatform Cryptographic Operations API

The GlobalPlatform API[13] 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 Physical Unclonable Function

Physical Unclonable Functions (PUFs), first described in 2002 [14], 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 REF. 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 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.

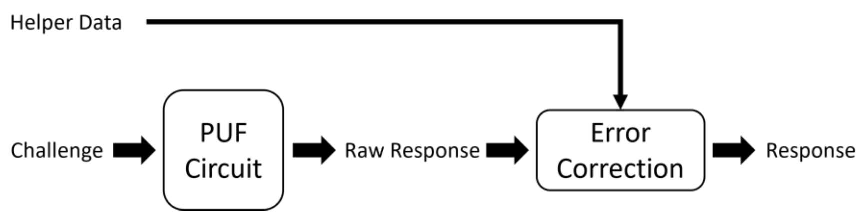

Figure 3: High-Level PUF Architecture

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 [15], 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 Integrateds ChipDNA [16], Pufsecurity's PUFrt [17] and PUFcc [18] or Synopsys Software-based PUF IP [19] these represent proprietary PUF solutions catering to diverse use cases. Notably, the vendors do not disclose an openly accessible PUF API. Therefore, one of the tasks for the next project period is to specify a unified API for PUFs used within CROSSCON.

3.2.3 Hardware Primitives for environmental Fingerprinting

Similar to Physical 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 Wireless Fidelity (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 Physical 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.

Consequently, a device must be equipped with a Wi-Fi chip to gather information about wireless transmissions in the environment. Further, the firmware of the Wi-Fi chip must support a so-called monitor mode that enables arbitrary monitoring of all traffic on any wireless channel. Additionally, a dedicated memory section must be allocated to the trusted service, which is utilized for storing and writing the collection of transmitter information. This area is accessible only by the relevant trusted service and is safeguarded against unauthorized access. Eventually, the collected information, resulting in the digital fingerprint of the location, needs to be transmitted to an external party via any form of end-to-end secured channel.

3.2.4 Description of API Demonstrator:

As part of the initial demonstrator showcasing the first results of CROSSCON WP4, our objective is to present an API demonstrator of the PSA Crypto API alongside the integration of cryptographic hardware accelerators. In this initial demonstration, we aim to achieve the following:

1. Implement an application capable of standard cryptographic operations (e.g., Encryption/Decryption, Hashing) using the PSA crypto API.
2. Showcase interoperability by running the demonstrators on both RISC-V and ARM devices.
3. Incorporate domain-specific hardware in the form of cryptographic hardware accelerators available on the devices and perform operations with and without hardware acceleration.

Currently, the demonstrator includes a ESP32-C3 (RV32IMC) and the LPC55S69 (ARMv8-M).

Moving forward in the project timeline, we intend to expand the demonstrator by defining interfaces for additional domain-specific hardware, such as PUFs or WiFi Channel State information. We will integrate these interfaces, along with the necessary interfaces for other CROSSCON trusted services, to establish a seamless interaction with domain-specific hardware for CROSSCON services.

The current version of the demonstrator can be found in Deliverable 4.2 (D4.2)[20].

4 Hardware-Accelerated Control Flow Attestation

Control-Flow Attestation (CFA) is a method employed for remotely ensuring the integrity of a program's execution. It encompasses the creation of a log detailing the instructions executed during the program's runtime, which is then utilized to validate the correct execution of the program. This technique serves to identify tampering or malicious activities, such as the insertion of harmful code or manipulation of input data. CFA can be realized through hardware-based solutions, software-based solutions, or a hybrid approaches.

4.1 System Overview

The process flow of our approach comprises both the Training pipeline, executed by the verifier, and the Inference stage, where attestation occurs. Initially, during Execution Tracing, the prover, represented in the diagram in Figure 4, gathers the execution trace consisting of basic block addresses and transmits it to the verifier. Subsequently, in the Preprocessing stage, the verifier converts the execution trace into a graph representation, termed as the execution graph, by extracting all pertinent features, facilitating subsequent stages.

In the Training stage, the verifier proceeds to train the Variational Graph Autoencoder (VGAE) model using the single execution trace received from the prover. Notably, the variable Z denotes the embedding of the input graph, while Val_set signifies a limited set of traces employed to determine the attestation threshold, denoted as t.

During the Inference stage, the prover submits an execution trace for attestation. At this step, the verifier leverages the trained encoder to extract the embeddings (Emb) of the execution trace, which are then compared with the embeddings (Emb_0) of the training graph to ascertain their similarity or dissimilarity via a distance measure denoted as d. Ultimately, attestation is performed through a threshold test.

In detail, the ML-based CFA scheme, illustrated in Figure 4, employs various techniques to conduct program execution attestation. Initially, execution traces of the program's execution are captured and transformed into graph representations. These traces, comprising a sequence of memory addresses traversed by the program counter during execution, are referred to as Exe_steps. In the preprocessing phase, the list of execution steps, Exe_steps = {s1, s2, ..., sn}, is converted into a graph representation. This involves constructing an edges list E = {e1, e2, ..., en}, where each edge ei = (vj , vk) signifies a connection between two nodes, vj and vk, belonging to V, the set of nodes in the graph. Each node in the graph represents a distinct memory address referenced in the execution steps.

Once the edges list E is established, it undergoes further preprocessing to convert it into graph connectivity in coordinate format (COO), referred to as A. This transformation is accomplished by applying the transpose operation A = E^T. This conversion is preferred as the COO format is more conducive for subsequent machine-learning steps. Additionally, while iterating on Exe_steps, the list of edges is constructed.

In the following, we detail the functionality of Trace Preprocessing, which will be implemented as a memory-mapped peripheral in CROSSCON SoC.

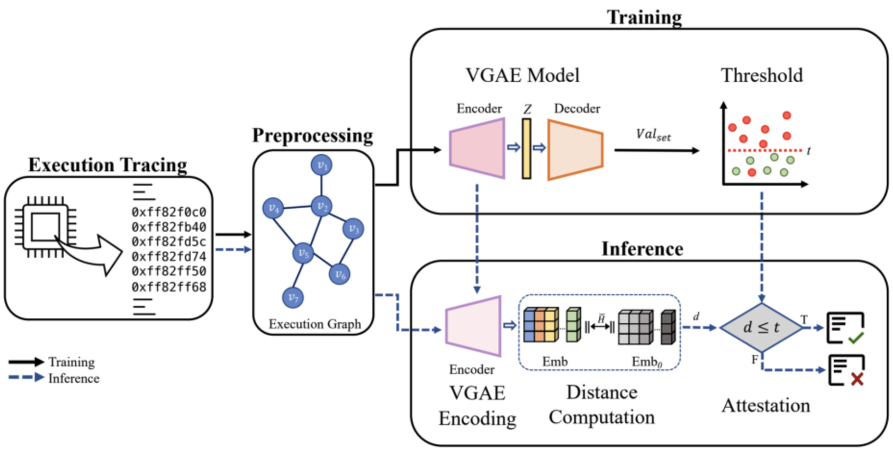

Figure 4: A comprehensive look at the CFA framework, encompassing both the training pipeline and the inference stage

4.2 Trace Preprocessing

During the feature extraction phase, our goal is to discern diverse attributes of each node v_i ∈ V within the graph G, which portrays the control flow of the software's execution. To accomplish this task, we leverage the output of a Tracer (e.g., TG RISC-V Nexus Trace1 or Arm's CoreSight), which furnishes a list of execution steps denoted by the memory addresses residing in the Program Counter (PC) of the CPU. Initially, we create the partial Control-Flow Graph (CFG) for a singular execution by compiling an edges list represented as an array E ∈ N^{n×2}, where n stands for the count of unique addresses (i.e., vertices) pertaining to a particular software. Given that we solely operate with PC addresses devoid of any additional information, we devised a feature extraction phase to unravel various characteristics of each node.

A key feature is the vertex degree, expressed as deg(v_i), representing the count of edges linked to v_i. This attribute offers insights into the node's connectivity by revealing how many edges are associated with it, which in turn can indicate the node's centrality within the graph.

Another important metric is the number of visits, indicating how frequently the corresponding address is accessed. This feature provides valuable information about a node's overall activity during execution and its visitation frequency.

Furthermore, we establish the initial and final visitations of each node, denoting the first and last instances the address is accessed throughout the execution steps. This attribute offers insights into the temporal usage pattern of a node, indicating its usage onset and conclusion.

Additionally, we compute the count of incoming and outgoing edges for each node. The former represents the number of edges terminating at v_i (i.e., the observed node), while the latter signifies the number of edges originating from v_i. These metrics offer valuable information about the control flow within the execution, revealing the volume of pathways entering or departing from a specific node.

Additionally, we calculate the visit frequency, defined as the ratio of visits to the total number of execution steps. Furthermore, we determine the time of use, which is the interval between the last and first visit, indicating the duration for which the node was utilized during the execution. These metrics offer insights into the relative significance of a node in the execution process and the duration of its utilization.

Moreover, we compute the standard deviation of all visits for a given node i, where the list of visits is represented as vi_si = {si, sj , ..., sk}, with sk being the k-th step where the address represented by vi is

1 https://github.com/riscv-non-isa/tg-nexus-trace

last seen. Additionally, we calculate the mean of the distances between consecutive visits, derived from the same list of visits, vi_si = {si, sj , ..., sk}, by computing the average of the differences between each pair of sequential visits, i.e., mean ((sj − si), ..., (sk − sj)). These metrics offer insights into the temporal patterns of visitations, indicating whether visits are clustered or distributed evenly over time.

Additionally, we calculate the average number of visits for the incoming and outgoing neighbors of all nodes that either have an edge ending at vi or are the endpoint of edges starting from v_i, respectively. These metrics offer insights into the activity levels of neighboring nodes, indicating the degree of activity within the surrounding nodes.

After the feature extraction process, we standardize them based on the length of the trace to ensure comparability across various execution traces. Additionally, we categorize these features according to their specific relevance within the context of our study and organize them into Table 1. It's important to emphasize that all extracted features are computationally efficient: we've estimated their algorithmic complexity to be approximately O(n), where n represents the length of the trace. This efficiency facilitates rapid data preprocessing and efficient runtime attestation, aligning perfectly with the objectives of our study.

Table 1: Synthetic list of extracted features

#	Feature
1	Vertex degree
2	Number of visits
3	First visit
4	Last visit
5	Incoming edges
6	Outgoing edges
7	Frequency of visits
8	Time of use
9	Std visits
10	Mean distance visits
11	Mean visits
12	Mean N. visits In. Neigh.
13	Mean N. visits Out. Neigh.
14	Mean last visit In. Neigh.
15	Mean last visit Out. Neigh.

In the following steps, we will implement the Trace processing as a hardware component that can be integrated within CROSSCON SoC for accelerating runtime attestation of applications running on a RISC-V core. Another option would be to integrate it as an FPGA-accelerated task on the ZCU102 to perform runtime attestation of applications running on the associated Arm Cortex-A53 core.

5 CROSSCON SoC

CROSSCON SoC is a an SoC design that provides secure RISC-V execution environment for mixed criticality IoT devices that require strong security guarantees, flexibility, small code size and low power consumption. CROSSCON SoC is based around Beyond Semiconductor's BA51 (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. Figure 5 shows the current fully featured high-level architecture of the CROSSCON SoC.

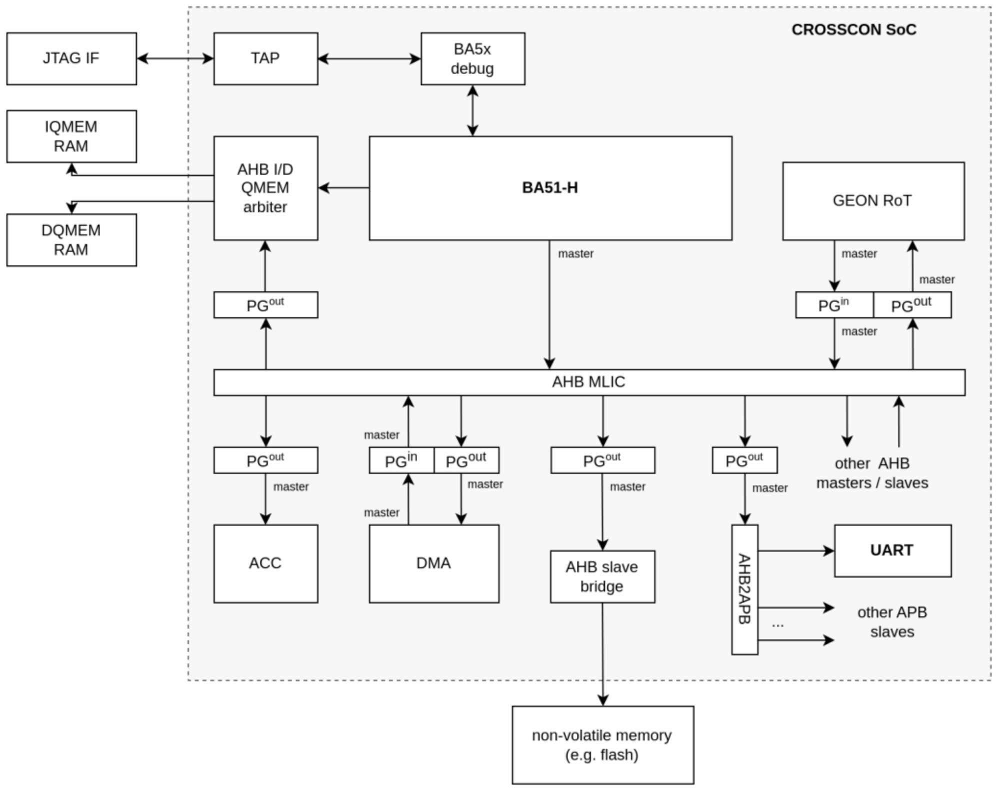

Figure 5: A high-level overview of the current architecture of the CROSSCON SoC.

The CROSSCON SoC has a single RISC-V 32-bit based processing core, called BA51-H, that has access to instruction and data RAM (instruction quick memory (IQMEM) RAM and data quick memory (DQMEM) RAM respectively), and several other HW modules connected through an AHB multi-layer interconnect (AHB MLIC). The modules connected to the AHB MLIC are: an (external) non-volatile storage (e.g. flash), GEON root of trust (GEON RoT), an accelerator (ACC) (e.g. AES/RSA accelerator), the AHB2ABP (AHB to advance peripheral bus (APB)) bridge, and the QMEM arbiter that allows other HW modules connected to AHB MLIC, as well as BA51-H, to access the instruction and data RAM. Furthermore, other HW modules can be added to the SoC by either connecting them through the AHB MLIC or the AHB2APB bridge, as, for example, the universal asynchronous receiver-transmitter (UART) module, that can be used by BA51-H to communicate with the outside world.

Note that the current architecture of the CROSSCON SoC described in this chapter might change as we refine the design of the SoC through the duration of the project. Furthermore, because the CROSSCON SoC is still being actively developed, not all functionality is implemented at this point in the project. The description and the bitstream of the current prototype can be found at [21]. We plan to add additional information regarding the architecture of the SoC to this chapter once it is further refined and finalized.

5.1 BA51-H core

The BA5x is a line of RISC-V cores provided by Beyond Semiconductor[22]. For the current version of the CROSSCON SoC, we have used BA51-H: an extended version of the BA51 core with additional virtualization support.

BA51 core: The BA51 (Figure 6) 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.

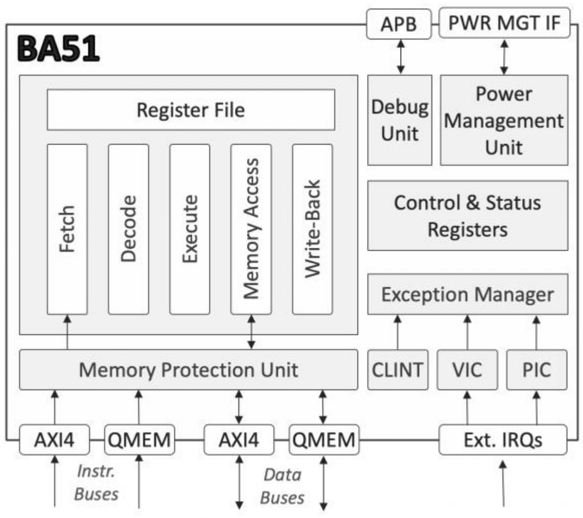

Figure 6: High-level architecture of the RISC-V BA51 core.

BA51-H: As part of the CROSSCON project, we have extended BA51 core with the 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 much more flexible because it allows one to establish several orthogonal TEE instead of using the idea of dual-worlds used in TrustZone. 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. When extending the BA51-H core, this were one of the main factors that we took into consideration. 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 2 shows the preliminary 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.

Table 2: The preliminary 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

From the preliminary 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 extended the Spike simulator[23], 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 extremely 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 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. The code of the extended Spike simulator can be found at [24].

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 [25] 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 [24] of the unified SPMP extension, and furthermore, implemented the extension in hardware as part of BA51-H [25] .

5.2 MLIC and access control

In the current architecture of the CROSSCON SoC, as shown by Figure 5, we intend to use the PG to control access to HW modules connected to the AHB MLIC on the level of maser and slave modules. By using PGs, we can precisely control which AHB master has access to which AHB slave, where this can be configured at runtime. By further extending the BA51-H core, we can get similar guarantees for VMs instantiated by the CROSSCON Hypervisor. This allows the software running in VMs to use HW modules without compromising isolation between VMs. We describe how the PG works in detail in chapter 6 and how we intend to use it in the CROSSCON SoC in chapter 6.7.

5.3 Secure boot

The SoC can be booted using the secure boot sub-system, called GEON RoT, which loads and authenticates an encrypted program stored in non-volatile storage (e.g., flash memory) into the instruction and data RAM. When the system is booted, BA51-H runs the first-stage code of the boot process stored in the ROM. The first-stage code configures the secure boot sub-system and provides it with the necessary secrets that are stored in the one-time programmable memory (OTP) and instructs the sub-system to decrypt, load, and authenticate the program stored in the non-volatile storage. BA5x then runs the program if the decryption and authentication are successful. The ROM and the OTP are a part of the RoT. How the RoT will be available inside of the CROSSCON SoC still needs to be determined.

5.4 Description of the demonstrator

For the initial demonstration of the CROSSCON SoC, we plan to provide a prototype of the CROSSCON SoC with a reduced set of HW features and a part of the supporting software that can be used to demonstrate certain aspects of SoC's functionality. As part of D4.2, we currently aim to include:

- An extended Spike simulator [24] used as a referenced execution environment that contains the prototype implementation of the unified SPMP model.
- A basic prototype version of CROSSCON SoC with BA51 core [21] without the necessary extensions to run CROSSCON Hypervisor, without GEON RoT, with a minimal set of HW modules (debugger, arbiter, UART, etc.) and without PG.
    - We plan to add missing HW features (BA5x core extensions, PG, GEON RoT, etc.) to the basic version of the CROSSCON SoC if time permits.

The description and the bitstream of the current prototype of the CROSSCON SoC can be found at [21].

6 Perimeter guard

In this section, we describe a HW module called Perimeter guard that can be used to share HW modules across different security domains of a system while preserving the isolation guarantees between the domains. We begin the chapter with an overview and motivation for the PG and continue with the description of its behaviour. The work in this chapter is a continuation of the work done as part of the task 2.3 (Hardware/Software Co-design) as described in CROSSCON Open Specification [26].

6.1 Motivation

A problem that we are addressing with PG is the following. Imagine that we have a SoC that divides its resources into multiple isolated domains (i.e. contexts or VMs). We want to integrate a new HW module into the SoC in such a way that it can be used by multiple domains but at the same time we want to preserve the isolation between the domains. For example, we want to be sure that a domain is not able to learn anything about other domains using the new module that we have added. It turns out, that there is no simple way of doing that for an arbitrary HW module because the modules are not usually designed with this in mind. One way that we could figure out if this is possible is to determine all the possible information flows through the module and find a way to use the module that satisfies our goal. 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 single domain (the safe way) or it is shared between domains, which can compromise the existing isolation (the unsafe way). To address this issue, we want a solution that:

- would allow us to share a HW module between domains without compromising the isolation,
- can be applied to existing HW modules, and
- does not require that we change the HW module in a major way.

In the following chapter, we describe one such solution, called perimeter guard (PG).

6.2 Overview

A *Perimeter guard* is a HW module that can be placed in front of a slave module on a bus to control access to the module according to the provided policy. For example, Figure 7 shows a setup where there are two masters, M1 and M2, that are connected to the HW module (MOD) through a PG over a bus where the PG's policy only allows M1 to access the module.

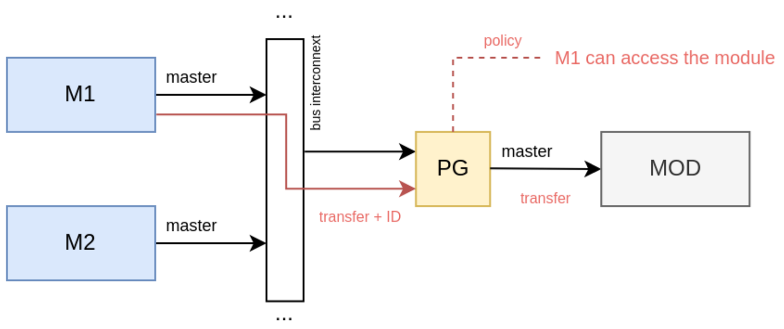

Figure 7: A basic SoC where we have two masters, M1 and M2, connected to a HW module (MOD) through a PG on a bus.

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

In order for the PG to decide which request can pass through, the master sending the request needs to provide the necessary information for the PG to make a decision. For example, this information can vary from a simple master ID, which identifies who send the request, to an MPU like behaviour where the request's address is also considered. (Figure 6 illustrates the former.) The information used to make a decision depends on the setup and the use case. Because of that, we do not limit the information that PG can use to decide when to allow access; we only require that the master can provide that information. How the information is provided to PG usually depends on the bus protocol that connects the HW modules.

We simplify further discussion by assuming that all the SW and HW resources of a SoC (e.g. code, data, threads of execution, computing cores, RAM, cryptographic accelerators, IO devices, etc.) are grouped into *domains*. A domain is a collection of SW and HW resources. For example, in the case of the CROSSCON Hypervisor, we could say that each VM is its own domain. Two domains are *isolated* if no information is shared between the resources belonging to those domains except through the allowed communication channels. Note that what belongs to a specific domain needs to be explicitly defined for each setup that we are considering. By having the idea of domains in mind, we can talk about using a PG to control access to a HW module from different domains by not losing any generality of our solution. We can always choose such domains so that we get appropriate granularity for a specific setup.

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

Figure 8 shows a simplified setup of CROSSCON SoC (Figure 5) where a PG is used to restrict access to HW accelerator (ACC). In this setup, we assume that each VM is part of a domain: VM1 is part of a domain D1 and VM2 is part of a domain D2. Each time a store / load instruction is performed in one of the VMs, the DID of the domain that the VM is a part of is passed along with the transfer that is performed on behalf of the store / load instruction. The PG can then decide for each transfer if it will pass it to the accelerator based on the transfer's DID.

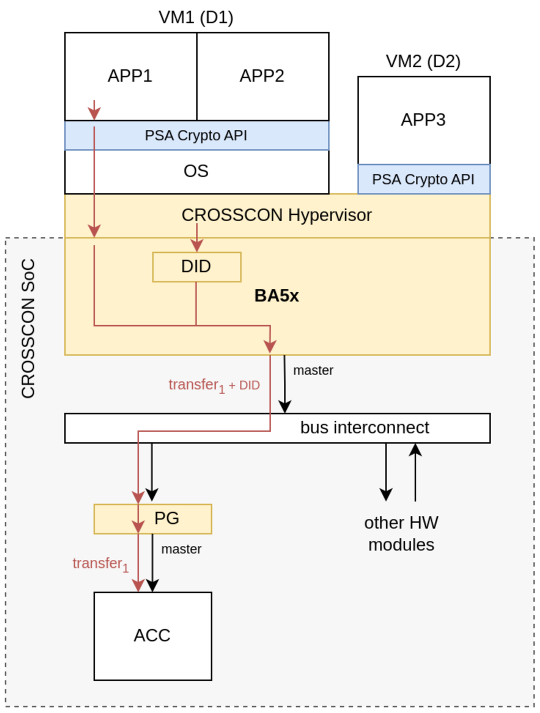

Figure 8: How a PG is used in the of CROSSCON SoC to restrict access to a HW accelerator.

For the initial implementation of the PG, we assume that masters and slaves communicate using AMBA 3 AHB-Lite protocol (version A) [27]. Where in the future, we also plan to consider other bus protocols. Furthermore, we assume that each transfer over an AHB bus is performed on behalf of a domain. Master transmits the DID alongside the transfer using a dedicated signal called HDID to let the slave know on behalf of which domain the transfer is performed. We also assume that all masters drive the HDID in an honest way, i.e. the HDID reflects the domain that caused the transfer.

The PG needs to be configured at the power on of the SoC (for example, at the boot stage of MCU) by providing the access policy (e.g. configuration) to the PG by the appropriate authority. How the PG can be configured is described in Annex I.

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

1. **Exclusive access mode** – Only one domain can access the module for the runtime of a system.
2. **Restricted address space mode** – Each domain has access only to a specific part of the address space of the module. The PG in this mode can be viewed as a module specific MPU.
3. **Time-sharing with reset mode** – A HW module is time-shared between domains. Every time another domain gets access to the module, the HW module is reset to a predefined state.
4. **Time-sharing with context switching mode** – A HW module is time-shared between domains. Every time the module is passed on to a different domain, the state of the module is stored and state belonging to the new domain is loaded into the module. (This approach is similar to context switching on MCU.)

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

- **External arbitration mode** – A PG allows an external master to decide who can access the HW module; the PG just enforces the decision. This is useful when we want to delegate the decision of who can access the HW module to external master.
- **Lock-release arbitration mode** – A PG provides a lock-release mechanism that allows a master to lock and release a module.

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

In the following chapter, we provide a threat model specific to the PG in the context of the CROSSCON SoC and describe the design and implementation of the PG in time-sharing with reset mode with external and lock-release arbitration. Where in the Annex I, we provide step-by-step, specification like, description of time-sharing mode with external arbiter. The detailed specification was prepared as an exercise in HW module design before the actual implementation to determine possible design and implementation problems. A more detailed explanation of how PG behaves in exclusive access mode, time-sharing with reset mode and time-sharing with context switching mode is described in chapter 6.2.3 (Perimeter Guard Architecture) of CROSSCON Open Specification – Draft [28].

6.3 The setup and the threat model

Following is a threat model specific to the problem we are trying to address with the PG. This section is a refined version of the threat model presented in D1.1 CROSSCON Open Specification – Draft[28] with additional details relevant to the CROSSCON SoC and the CROSSCON Hypervisor.

Setup: We assume a similar architecture of the CROSSCON SoC as sown in the Figure 5 without PGs. We leverage the BA51-H core and the CROSSCON Hypervisor, to establish several isolated VM that are allowed to interact between each other according to the configuration of the CROSSCON Hypervisor. We consider each VM to be in its own separate domain. Furthermore, we assume that each of the masters connected to the AHB MLIC is a part of some domain and is able to drive the HDID signal for each of transfers that it initiates.

Goal: Our goal is to allow a HW module to be added to the CROSSCON SoC in such a way that the module can be shared between domains (i.e. VMs and masters) while preserving the isolation between them.

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

We **trust** the isolation between VMs, the privileged domain configuring the PG, and the PG itself. We assume all the components we trust are functionally correct, have no bugs, and behave as expected.

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

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

6.4 Threat analysis

In order to preserve the isolation between VMs, we need to ensure that no information flows from one domain to another if that is not allowed. In general, we identified the following ways of how information can flow from one domain to another: (1) through the (internal) state of the module, and (2) through module's availability (i.e., when and how long the module is used).

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

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

6.5 Implementation and design: Time-sharing with reset mode

We can share a HW module by allowing the module to be used by several domains for different periods of time where only one domain has access to the module at a time. We call this approach *time-sharing*. This is a usual way of how HW resources are shared, as, for example, a computing core can be shared between different programs where only one program can run on the core at the time. For the initial version of the PG, we have decided to support time-sharing of modules.

When designing a HW mechanism for time-sharing modules, we need to consider several aspects of the design. First, we need to make sure that the mechanism preserves the isolation between the domains using the module. Second, we need to choose a scheduling mechanism that decides for how long and how often a domain will have access to the module. And third, we need to be sure that the mechanism can be efficiently implemented. We have already considered some of this design choices as part of the CROSSCON Open Specification - Draft[28].

In the CROSSCON SoC, HW modules communicate between each other using AHB bus protocol. For the initial version of the PG, we assume that they use the AMBA 3 AHB-Lite version A [27]. The role of the PG on AHB bus level is to restrict access to the HW module by filtering out transfers. The PG decides if a transfer should be passed-through or filtered out based on the DID of the master sending the transfer, which is provided by the master along each transfer by driving the HDID signal.

For further discussion, we say that a PG has two sides: the side connected to the interconnect, called *external side* (ES), and the side connected to the accelerator, called the *internal side* (IS). We often refer to sides when we need to clarify which signal has to be driven. Furthermore, we say that the master on the ES side is *ES master* and the slave on the IS side is the *IS slave*.

Figure 9 shows the initial design of the PG. We divide the design into several submodules that handle specific parts of PG's behavior: (1) filters that are responsible for filtering different AHB signals (*AHB address and configuration signals filter (AHB ACS filter), AHB HWDATA filter* and *AHB response filter*); (2) *slave reset logic* responsible for resetting the slave, (3) *error response logic* responsible to signal an error over the interrupt or the AHB bus; and (4) *byd_ahb_pg_cfg* logic responsible for providing the configuration interface.

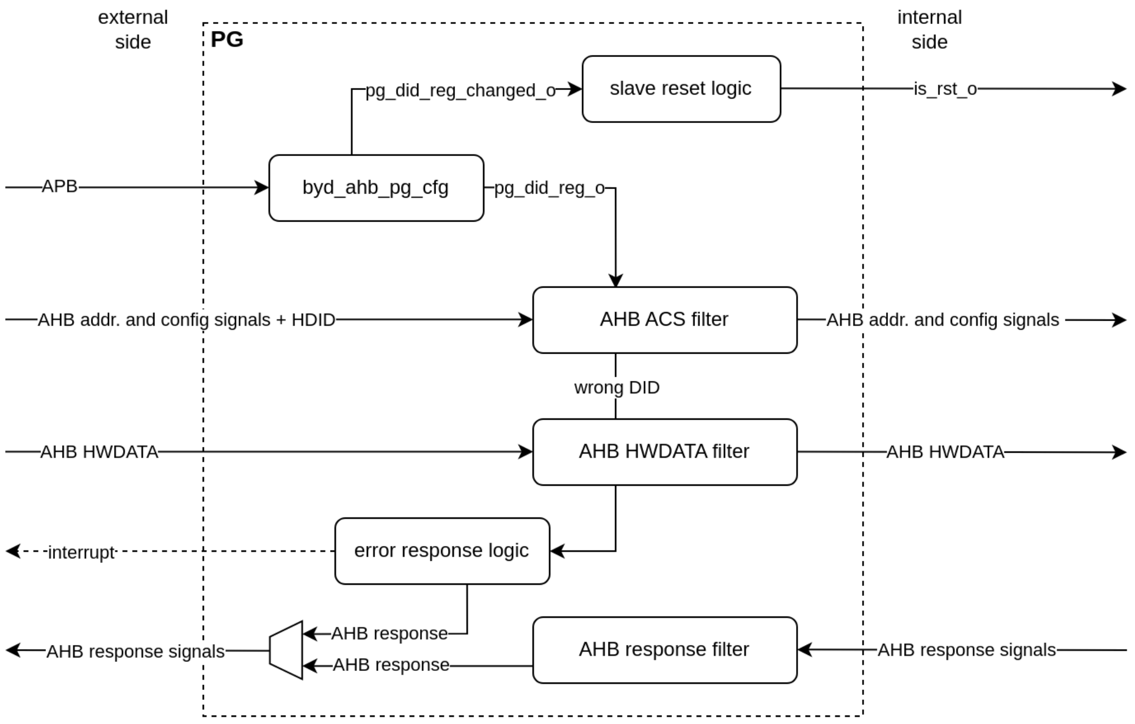

Figure 9: High-level overview of the design of a PG in time-sharing with reset mode

Signal filtering: AHB protocol divides each transfer into two phases (*address phase* and *data phase*) which are performed one after another in a way that the address phase of the next transfer is signalled at the same time as the data phase of the current transfer (i.e. transfers are *interleaved*). This requires us to handle two separate sets of signals, each related to a different transfer and potentially a different domain, in a single clock cycle. AHB protocol further supports burst operations where several related transfers are sent one after another. Additionally, each signal of a transfer is driven either by ES master or IS slave where you need to treat the IS and ES signals separately when filtering. All of this creates a situation where you need track several different context and conditions to filter out signals appropriately and prevent any unwanted information flows between domains. Any mistake in filtering can cause the entire bust to halt. If, for example, we filter out a response to a transfer, this can cause the entire bus to halt because the master that initiated the transfer will wait indefinitely for the response. This can cause the entire SoC and BA51-H to halt which can only be fixed by rebooting the SoC. From all of this, it follows that AHB protocol is fairly complex and filtering out transfers is not easy. We manage AHB's complexity by dividing the logic that handles filtering into several submodules. This allows us to have a smaller context that are easier to track and allows us to have simpler conditions for when to filter a specific signal. For precise conditions when a signal is filtered out, see the specification in Annex I.

Configuration: A PG can be configured through an APB interface which is driven by BA51-H. Currently, APB interface in external arbitration mode exposes the following registers (Table 3).

Table 3: Configuration registers of a PG in time-sharing with reset and external arbitration mode

Register	Size (in bits)	Address [8:2] (in hex)	Description
pg_did_reg	32	00	A register containing the ID of a domain that can access the IS slave. Used only in the external arbitration mode.
pg_cfg_reg	32	01	A configuration register where:
pg_cfg_reg[0]	1		- report_error_over_bus_bit - When set to 1, an error is signalled over AHB bust by raising the es_ahb_hresp_o signal.
pg_cfg_reg[1]	1		- report_error_over_interrupt_bit - When set to 1, an error is signalled over the interrupt wire.
pg_cfg_reg[2:31]	30		- Bits from 2 to 31 are reserved.
pg_interrupt_cause_reg	32	02	(Read only) A register that contains the DID of the last domain that caused an interrupt/error.

External arbitration mode: In external arbitration mode, the PG allows an external master to decide which domain has access to the module at a specific time. This can be done through the APB master configuration interface by setting the pg_did_reg register. The DID in the pg_did_reg determines the current domain that can access the module. In each of the filters, the HDID signal is compared with the pg_did_reg to decide if a signal can be passed through or it should be filtered out. When a signal is filtered out, the PG drives it to a default value.

The external arbitration mode is useful because it allows us to move the scheduling and arbitration logic into software, for example, scheduling could be done by a trusted application. This is especially useful for prototyping new scheduling algorithms because we can first try them out in SW before implementing them in HW.

Lock-release arbitration mode: The second arbitration mode that is also supported by the current implementation of the PG is the lock-release mechanism, which is a similar mechanism as mutex synchronization primitive in software. In this mode, the PG exposes additional configuration registers, pg_lock_reg and pg_read_reg, over the AHB interface that allows a master to lock and release the IS slave. When a master wants to use the IS slave, it first needs to lock it by reading from pg_lock_reg register. If read returns one, the IS slave was locked successfully and the master can use the slave. When a master has finished using the IS slave, it can release the slave by reading from pg_read_reg register. If one is returned, the slave was freed and another master can lock it. The lock-release arbitration mode is simple to implement and use because the arbitration logic is fairly simple.

Threat analysis: In the case of the time-sharing with reset mode, we reset the state of the module when passing the module from one domain to another. This way no information from the previous domain is left in the module's state, which prevents interference [29] in both directions: from the previous domain to the new domain and vice versa. By resetting the module when passing the module from one domain to another, we prevent all state related information flows between the domains.

Regarding timing attacks, the external arbitration mode does not provide any solution for timing attacks directly because the arbiter has full control of which domain can access the module. This can be addressed by the arbiter using a fair scheduling mechanism that hides timing information, for example, round-robin scheduling with timeslots of the same size where time slot is always given to a domain even though it might not need it. The same does not hold for the lock-release arbitration mode. In the lock-release arbitration mode, (untrusted) master can use the module in a first com fist serve manner which can be used to obtain some information via timing information flows.

The time-sharing with reset mode is only applicable to HW modules that can be reset to a predefined state. We expect that this usually can be done for HW accelerators (e.g. ACC, RSA, ...) with no or minimal modifications to the module, but it might be harder to do for other modules, as, for example, peripheral devices (e.g. I/O devices).

Resetting a module can be problematic for the application that uses it. For example, in the case of an external arbiter, the scheduling mechanism might not know when the IS slave has completed some operation. If the module is reset during the operation, it loses its progress. To handle the loss of information / state of the module, the application (or intermediate software) needs to be aware that the reset happened, which means that the application needs to be designed with this in mind. Furthermore, it's hard to determine for how long a domain should have access to the module. From this it follows that the time-sharing with reset and external arbitration mode is not an adequate solution for mitigating time-sharing attacks and does impact the application using the module in a severe way. Similar holds for the lock-release arbitration mode. But in the lock-release mode, the master that uses the module has full control of when to release the module, which allows him to wait until the operation finishes and store the module's state if needed. The downside of this approach is that the master can decide to never release the module, which leads to a denial-of-service attack.

In order to address timing attacks, loss of state and denial-of-service attacks, we need to find a more appropriate behaviour for the PG that avoids these issues. One such behaviour could be time-sharing with context-switching as we described it in chapter 6.2.3.3 (PG Operation Mode: Time-sharing with Context-switching) of CROSSCON Open Specification – Draft [30].

You can find additional information about PG and how it behaves in Annex I.

6.6 Software interface

We want the application running inside of a VM to be able to use the HW module protected by a PG in a seamless way. With that in mind, some of the properties that the interface needs to have are:

- the interface needs to be easy to use, e.g.
  - it hides as much complexity as possible,
  - it hides the details specific to the HW module, where ideally the user does not know that the HW module is used;
- if possible, it hides the fact that the HW module is shared between VMs; and
- it guarantees caller isolation between VMs.

How this can be achieved highly depends on the PG's behaviour and how we want the basic interaction between SW and the HW module to look like. We have provided an example of how this can be done for a cryptographic accelerator in chapter 6.2.4 (Integration with the CROSSCON SoC) of D2.1 CROSSCON Open Specification - Draft[30]. Finding the right interface design that provides the wanted guarantees is an active area of work that we plan to follow as part of tasks T3.2, T4.1 and T4.2.

Caller isolation: We can use the PG with the BA51-H core and the CROSSCON Hypervisor to obtain cross VM caller isolation through the entire HW/SW stack, if the PG guarantees noninterference between VMs. For example, if we assume that (1) the HW accelerator protected by the PG is a cryptographic accelerator, (2) that the PG is in time-sharing with reset mode, and that (3) the cryptographic accelerator can be used through PSA Crypto API, we can provide caller isolation for the PSA Crypto API available in each of the VMs. When an application inside of the VM calls the PSA Crypto API and wants to encrypt / decrypt some data, the instructions of the call perform several load / store requests that store or load some data to the cryptographic accelerator. If the cryptographic accelerator is assigned to the VM where the calling applications runs in, the load / store request will be passed through to the accelerator; otherwise, the request will be rejected by the PG. If the PG guarantees noninterference between VMs, we can be sure that no other application running in other VMs can obtain information about the caller. Furthermore, this also holds for other masters on the bus that are not a part of the same domain as the VM to which the accelerator is currently assigned to.

6.7 The PG setup in the CROSSCON SoC

The BA51-H core has an additional register through which the CROSSCON Hypervisor can let the BA51-H know which VM is currently using the core. This allows the core to drive the HDID signals for each transfer, but other masters usually do not have this information and thus do not know to which domain they belong to. This can be solved by introducing a HW module called PG^{in} that can be placed in front of an arbitrary master to drive the HDID. Which DID is assigned to a master depends on how it is configured by the CROSSCON Hypervisor or trusted service at boot. Note that configuring the DID of the PG^{in} determines to which domain the master belongs to. The precise functionality of PG^{in} still needs to be determined.

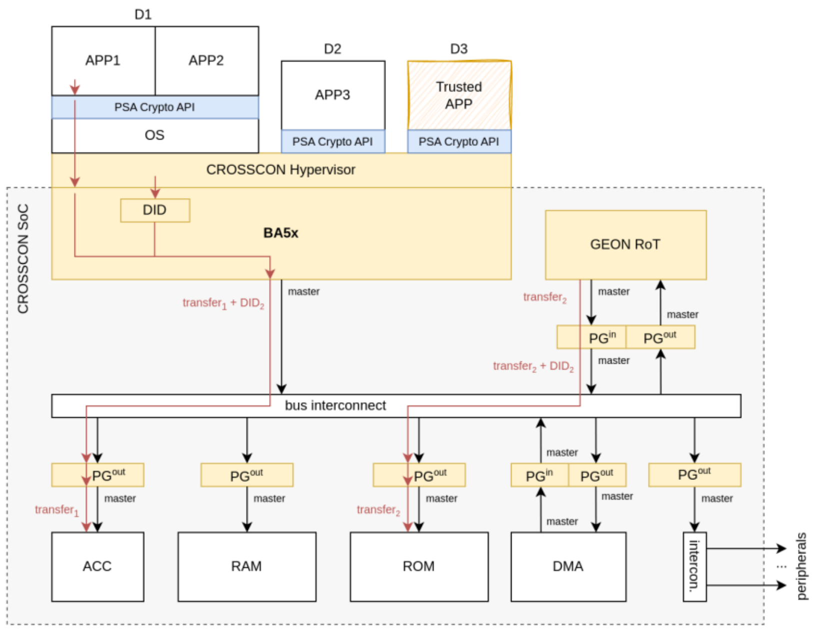

Figure 10: A CROSSCON SoC setup where each HW module, except the BA51-H, is connected to the bust interconnect through a PG^{in} or a PG^{out}.

Figure 10 shows a CROSSCON SoC setup where each master on a bus, except the BA51-H core, has a PG^{in} module that drives HDID for each transfer. To avoid confusion, all the PGs are renamed to PG^{out}. By using PG^{in} and PG^{out}, we can construct a bus interconnect where all the transfers over the interconnect are marked with DID of the domain that caused the transfer. This allows us to obtain interesting security guarantees, as, for example, being sure that only HW modules that are a part of the same domain can communicate between each other.

7 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 FPGAs (vFPGAs) for efficient resource utilization and enhanced security. Then, we address the critical issue of Intellectual Property (IP) protection, proposing solutions such as hardware-based trusted FPGA shells and software-based trusted FPGA services. 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. Moreover, we emphasise the importance of bitstream verification to mitigate potential security risks associated with malicious hardware designs. Overall, in the following, we provide insights into FPGA provisioning strategies and measures essential for ensuring the security of FPGA-based systems in diverse computing environments.

7.1 Background on FPGA

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 tables (LUTs) and flip-flops. Configurable logic blocks are connected together or to I/O blocks through programmable routing interconnects. Other components of FPGAs are memory blocks (BRAM) and digital signal processing blocks (DSPs) 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, block RAMs, 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, utilising it to program the configuration memory. This memory then dictates the behaviour 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 [31].

7.2 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 [32], 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 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 [33][34][35][36]. 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 these logically isolated regions as virtual FPGAs (vFPGAs). Subsequent sections will explore FPGA partitioning options and best practices for creating virtual FPGAs.

7.3 Partitioning of the FPGA

In addition to the vFPGAs, a dedicated area within the FPGA is allocated for the FPGA shell. This shell is crucial in simplifying interactions with FPGA-specific interfaces and pins. Essentially, the shell is a hardware circuit that manages the flow of information within the FPGA, ensuring smooth communication between the FPGA fabric and external devices. The key functionalities it provides include:

- Supplying clock signals to vFPGAs.
- Facilitating 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.
- Managing data transfers between the vFPGAs and external memory.
- Optionally supporting the configuration of vFPGA resources via an internal interface.

Furthermore, the shell may incorporate security features such as access control and encryption/decryption to bolster the protection of sensitive data and resources. In addition, the shell can be engineered to include a fence of empty logic to enforce isolation between hardware circuits on vFPGAs (as shown in Figure 12). Figure 11 demonstrates a simple design of the FPGA shell.

By providing essential control logic and I/O interfaces, the shell ensures a user-friendly interface for seamless data interactions. Various virtualisation schemes have proposed for partitioning a single physical FPGA into multiple vFPGAs [34][35][36][37].

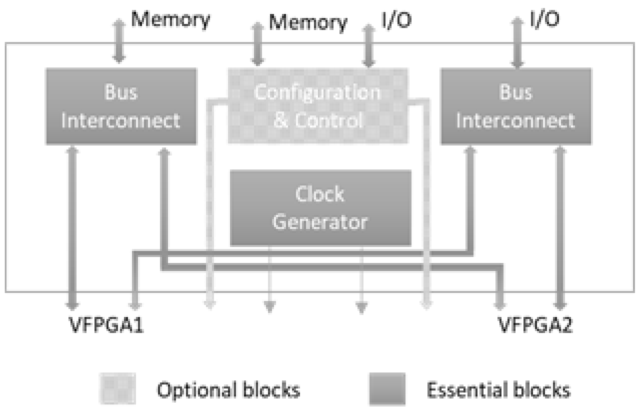

Figure 11: FPGA Shell Design

7.3.1 Fixed-Size Virtual FPGAs

Fixed-size virtual FPGAs capitalise on the partial reconfiguration feature, wherein the system administrator predetermines the maximum count of partially reconfigurable regions along with their respective boundaries [34]. This strategy entails delineating the dimensions and boundaries of each region and establishing static empty logic fences between them to ensure isolation. Provided with comprehensive information on the available partially reconfigurable regions, i.e., vFPGAs, their precise locations, and boundaries within the FPGA, hardware circuits intended for execution on the FPGA can be synthesised or compiled to operate on any designated region. Consequently, a distinct partial bitstream can be generated for each hardware circuit and region.

During runtime, a scheduler evaluates the availability of virtual FPGAs and allocates one accordingly. Then, the corresponding hardware circuit's bitstream is partially configured on the FPGA without disrupting the operations of other regions. Figure 12 illustrates a sample scenario wherein an FPGA is partitioned into two fixed-size vFPGAs, each independently manageable. The benefits of this method include streamlined FPGA resource management, a simplified FPGA shell design, and fast response times for serving FPGA requests, as the scheduler only needs to verify the availability of reconfigurable regions rather than querying and estimating the availability of FPGA's resources in terms of primitives.

It's noteworthy that since FPGA bitstreams for hardware circuits or designs are pre-generated, they can be promptly utilised to partially configure the allocated region. However, a drawback of this approach is its inability to exploit FPGA resources fully; thus, the available region might either exceed or fall short of the hardware circuit's requirements, necessitating potential partitioning across multiple vFPGAs. Although this partitioning is not permanent and can be altered at each FPGA power cycle, it mandates repeating the entire place-and-route and bitstream generation processes.

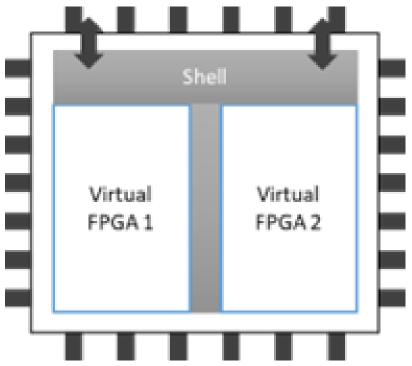

Figure 12: Example of FPGA with two fixed-size virtual FPGAs

7.3.2 Dynamically Sized Virtual FPGAs

Dynamically sized regions offer increased flexibility in adjusting the size of virtual FPGAs, enhancing adaptability to varying computational demands. Beyond the FPGA shell, the remaining FPGA resources collectively constitute a global, partially reconfigurable region available for runtime allocation to fulfil user requests [35]. This global area progressively accommodates hardware circuits, capitalizing on the hierarchical partial reconfiguration feature in modern FPGAs, which allows the splitting of a partially reconfigurable region into sub-regions. In this scenario, it is assumed that hardware designs that will run on the FPGA are synthesized into their corresponding netlists (and list of their requirements), and at least one default bitstream for each design is generated for a default location on the FPGA. Upon receiving a request to run a hardware circuit on the FPGA, a zone manager, a software component responsible for FPGA resources management in the system, assesses the availability of FPGA resources within the global area. If sufficient resources are available to fulfil the hardware design requirements, the default bitstream of the intended design partially configures only the necessary FPGA resources within the global area, leaving the surplus resources unaltered. Figure 13 shows an FPGA with two dynamically sized vFPGAs that are configured using their default partial bitstreams.

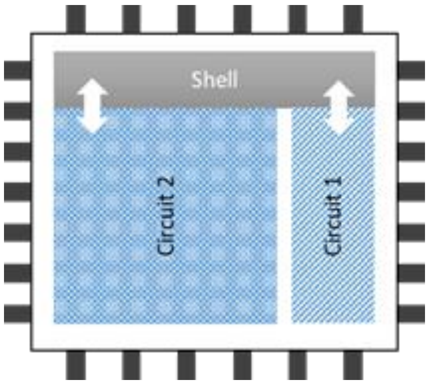

Figure 13: Dynamically sized vFPGAs with two circuits configured using their default bitstreams

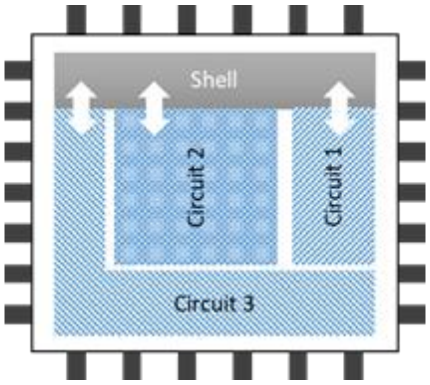

Figure 14: Dynamically sized vFPGAs with three circuits configured after retargeting them

However, in scenarios where a default bitstream of the desired hardware circuit is unavailable or does not match the allocated FPGA resources, the processes of place-and-route and bitstream generation are reiterated, utilising the actual available FPGA resources. Figure 14 shows a retargeting process, where the circuits 1 and 2 are pre-emptively interrupted to route-and-place the three circuits together and generate a single bitstream that configures the three circuits. To prevent data loss, due to interrupts, circuits have a special signal that can be asserted to allow the circuits to save data before interruption. This iterative process, conducted at runtime, incurs a performance overhead. When subsequent requests cannot be accommodated due to resource constraints within the global area, the zone manager re-evaluates the requirements of all active circuits. If a combination of the resource demands of these circuits can be accommodated within the FPGA, a retargeting process is performed in which the existing FPGA resources are reclaimed. Subsequently, a new round of place-and-route and bitstream generation processes is initiated, consolidating all hardware designs into a single bitstream to configure the FPGA.

While this method offers the potential for optimal flexibility and utilisation of FPGA resources, it also introduces significant overhead in terms of both management and performance. The intricate nature of this approach stems from the necessity for meticulous management of FPGA resources to ensure optimal utilisation while enabling the concurrent execution of multiple hardware designs without interference. The incorporation of dynamic retargeting further amplifies the complexity of the system, necessitating the seamless integration of diverse hardware designs and the repetition of steps within the FPGA development flow. Moreover, this approach demands the implementation of a sophisticated communication and memory management infrastructure within the FPGA to effectively accommodate the varying number of hardware designs concurrently running on the FPGA.

7.4 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.

Recent research [38, 39, 40, 41, 42, 43] has highlighted potential security risks using malicious FPGA designs. Remote physical attacks become feasible, including denial-of-service attacks on FPGAs, fault injection, or side/covert channel attacks. The objective of such attacks is to exploit power, timing, or thermal irregularities in neighbouring computing resources (CPUs, GPUs, or FPGAs) that share the power supply or distribution network with the malicious circuit on the FPGA, with the goal of extracting sensitive data. Therefore, ensuring the trustworthiness of hardware designs is paramount to safeguarding against these potential security risks. Nevertheless, we consider physical attacks on the FPGA that require direct physical access or close proximity to mount probes on or near the device to measure physical aspects of the device 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.

7.5 IP Protection

Partial bitstreams, which embed IP HW 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 [44] 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, which may not align with the current deployment model of commodity FPGAs. To address this challenge, two alternative approaches can be used, each with its underlying assumptions and prerequisites:

- Using a hardware-based trusted FPGA shell. Leveraging an internal configuration port of the configuration engine on the FPGA, a partial bitstream can be configured internally using the static region, specifically the FPGA shell in our context. In the process, the shell is responsible for decrypting and verifying the integrity of the partial bitstream. Initially, the encrypted partial bitstream is transmitted to the shell, where it undergoes decryption and verification. Subsequently, the validated partial bitstream is configured onto the designated partially reconfigurable region through the internal configuration port to the configuration engine. External FPGA configuration ports can be disabled by the trusted shell to prevent unauthorised access to the FPGA's configuration memory. This would thwart any attempts to read out IP bitstreams. It is imperative to emphasise that ensuring the security of the FPGA shell is paramount. This can be achieved by employing cryptographic solutions embedded within the FPGA, i.e., the FPGA owner's keys.

- Using a trusted application TA or VM that runs in TEE and exclusively controls FPGA services. The partial bitstream can be decrypted, verified, and sent to one of the external[2] 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. Note that this might not be supported in commodity FPGAs.

To enable IP protection, a key exchange protocol between the application requesting the FPGA services, and the trusted shell can used to share a secret key. The secret key is utilised to encrypt either the partial bitstream itself or the user's secret key, which is used for encrypting the partial bitstream. Furthermore, it is worth noting that a user may request to attest the FPGA shell [45] or the configuring enclave to confirm their integrity, further enhancing the security measures.

7.6 Bitstream Verification

Recent research has highlighted the potential for malicious hardware circuits on FPGAs to launch remote physical attacks, such as Denial of Service (DoS) and side-channel attacks[38][39][40][41][42][43][46]. Malicious circuits configured on the FPGA can be implemented to draw excessive current from the FPGA's power supply, causing the entire platform to cease functioning. Additionally, digital sensors can be configured on the FPGA to exploit phenomena like crosstalk between wires, voltage fluctuations in shared power supplies, or thermal variations, enabling the inference of sensitive data from neighbouring hardware identification circuits.

Given these threats, pre-emptive detection of circuits with sensor or power-draining capabilities is crucial prior to their deployment on the FPGA. Academic research suggests the use of virus scanners tailored for FPGA configuration bitstreams [47][48]. This seminal approach requires continuous update of the malicious circuits' signatures in the database. Further, the target specific FPGA families. However, any detection mechanism, whether commercial or academic, necessitates access to either the bitstream or the netlist of the hardware circuit that the client intends to upload. This presents a challenge in scenarios where intellectual property (IP) protection mandates the use of encrypted bitstreams. To reconcile the need for IP protection with the necessity of malicious circuit detection, an FPGA virus scanner can operate within an enclave. Upon completion of the scanning process, an authenticated report detailing the status of the scrutinised hardware IP bitstream, along with its checksum, is generated within the enclave and stored alongside the encrypted bitstream.

This approach enables the scanning of hardware IP bitstreams while maintaining encryption. Users can verify the integrity of the enclave housing the virus scanner to ensure the security and authenticity of the scanning process, safeguarding against potential leakage or manipulation of the bitstream. The scanning step is discretionary, particularly in cases where the hardware IP design originates from an untrusted source.

7.7 Workflow

Before deploying FPGAs, the system administrator initiates critical decisions regarding FPGA partitioning, encompassing the choice between dynamically sized or fixed-size virtual FPGAs (vFPGAs) and the selection of communication interfaces within the FPGA shell. These decisions are driven by the capabilities and features of the FPGA device and the specific requirements of the hosting platform.

The subsequent step involves the design and implementation of the FPGA shell on the static region of the FPGA by the system administrator, alongside clearing and blanking partial bitstreams for the vFPGAs. Variants of the FPGA shell can be tailored to perform different functionalities, including decryption, verification, and configuration of partial bitstreams. Moreover, the shell ensures the security of the user's IP by preventing unauthorised access to the FPGA's configuration memory

[2] External to FPGA fabric.

through external ports. Note that access to the FPGA shell and its memory region is restricted to CROSSCON Hypervisor or to a dedicated trusted application (TA) or virtual machine (VM) reserved for managing FPGA resources, which we refer to as FPGA trust anchor (TA_{FPGA}), see Figure 1. TA_{FPGA} can take over the decryption and verification of partial bitstreams before their configuration. Otherwise, the FPGA shell shall perform this process. Both TA_{FPGA} and FPGA shell can be attested.

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

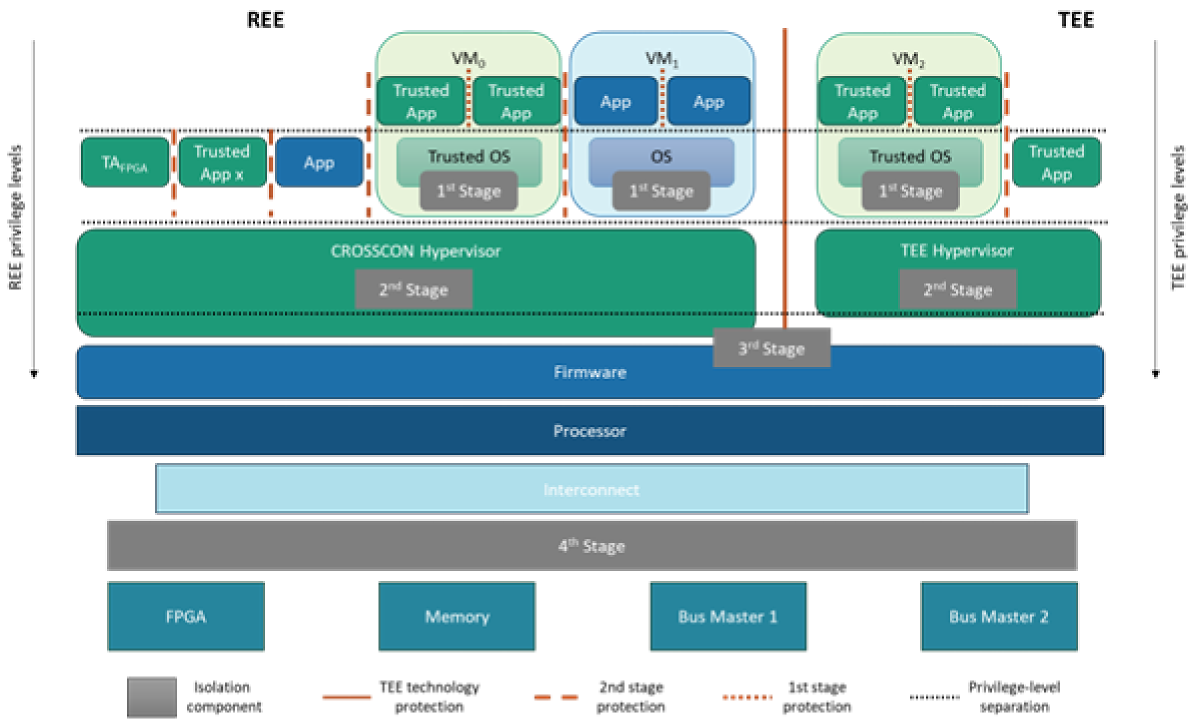

Figure 15: CROSSCON Stack

Users intending to deploy their IP designs must meet the defined physical constraints to place their designs within the boundaries of vFPGAs. However, users also have the flexibility to provide multiple instances of a task, each targeting a different vFPGA. This flexibility enhances the likelihood of successful deployment on the FPGA. Assuming a trusted application or virtual machine is requesting to run a task on the FPGA, the following steps will be taken:

1. vFPGA allocation: The process is initiated with a request from TA/VM to execute a task on the FPGA, specifying the location, count and sizes[3] of the task's partial bitstreams (i.e., FPGA configuration files) stored on disk. Given that TA_{FPGA} is available, vFPGA allocation requests are directed to TA_{FPGA}. Subsequently, the CROSSCON Hypervisor/TA_{FPGA} verifies the availability of a vFPGA and its corresponding partial bitstream. If both are available, CROSSCON Hypervisor/TA_{FPGA} assigns the vFPGA and provides the requesting TA with the FPGA shell/ TA_{FPGA} public key.

[3] Partial bitstream size depends on many factors including the vFPGA size, therefore one or more sizes can be passed in the request.

2. Secret key passing: Following vFPGA allocation, the TA uses the public key to encrypt the secret key used for encrypting and signing the task's partial bitstream. The encrypted secret key is then sent to CROSSCON Hypervisor/TA_{FPGA}. If decryption and verification occur in the TA_{FPGA}, the TA_{FPGA} loads the partial bitstream in the shared memory with the FPGA shell and instructs it to start the configuration process. Otherwise, the encrypted secret key and the encrypted partial bitstream are loaded on the shared memory region for the FPGA shell to decrypt and configure.
3. vFPGA configuration: The FPGA shell configures the partial bitstream on the allocated vFPGA. Upon successful configuration, the FPGA shell confirms the completion of the process.
4. vFPGA access control: Access control rules for the accelerator can be set using the requesting TA/VM ID. Consequently, the task is ready for execution. Note that access control to accelerators on vFPGAs can be achieved through (i) TA_{FPGA}, which can control access to vFPGAs when communication between the allocated vFPGA and the requested TA/VM occurs through TA_{FPGA} or (ii) isolation components, such MMUs and IOMMUs, as shown in Figure 15
5. vFPGA deallocation: Since FPGAs do not support pre-emption, it is left to the system administrator to decide whether a vFPGA is allocated for a specific task until the task finishes or allocated for a time slot (fixed or variable). The system administrator makes decisions regarding the duration of vFPGA allocation until a task is completed or for a predetermined time slot (fixed or variable). Consequently, the TA must be able to terminate a task by sending a request to CROSSCON Hypervisor/TA_{FPGA}. CROSSCON Hypervisor/TA_{FPGA} can also send a termination request to the TA to allow it to copy or delete sensitive data before terminating the task on the vFPGA. This ensures efficient resource utilisation and effective management of FPGA resources based on task priorities and system requirements. Note that upon vFPGA deallocation, a special partial bitstream is configured to erase the previous configuration.

7.8 Description of the Demonstrator

Our work will be 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), an FPGA fabric by AMD, among others [49]. The platform supports ARM TrustZone technology. The FPGA will be partitioned into three fixed-size regions: a static region where the FPGA shell is implemented and two vFPGAs that can be allocated to users. The FPGA shell will implement the communication and clock management to the vFPGAs using Xilinx IP cores. Each vFPGA will have a dedicated I/O interface such that vFPGAs can be controlled by different masters.

In the demonstrator, we opt for using TA_{FPGA} for managing and allocating vFPGAs according to the workflow described above. TA_{FPGA} will run on top of a Xilinx MicroBlaze microcontroller, a softcore implemented as part of the FPGA shell that has exclusive access to the internal configuration port. The shell is presumed responsible for configuring vFPGAs internally by utilising the internal configuration port of the FPGA configuration engine. TA_{FPGA} sends a partial bitstream to the FPGA shell by loading it to a pre-defined shared memory region with the FPGA shell. The partial bitstream is then read by the FPGA shell and forwarded to the internal configuration port of the FPGA configuration engine.

To safeguard the FPGA shell bitstream, cryptographic keys can be programmed onto the FPGA, enabling the protection of the shell's bitstream. Note that if protecting the confidentiality of the FPGA shell is unnecessary, the system administrator may opt to protect its integrity only, particularly if the shell design does not contain sensitive data or keys. Secure boot process capabilities with the ZCU102 can be leveraged to load the shell on the FPGA.

In the next steps of the demonstrator, we will extend the functionality of TA_{FPGA} to include the cryptographic operations. Then, we will integrate TA_{FPGA} within CROOSCON according to one of the two options: 1) a Guest VM hosting TA_{FPGA}, where CROSSCON Hypervisor itself ensures that access to the FPGA resources occurs only for the specific VM hosting the trusted service (Figure 16, Figure 17), or 2) Fully implemented on TrustZone (Figure 16), where TA$_{FPGA}$ providing FPGA services is hosted on a Trusted OS, e.g., OPTEE, running alongside a general-purpose OS, and no hypervisor intervenes. Access control will be to FPGA resources will be enforced based on solutions available by Xilinx [50].The implementation is performed using a suite of AMD Xilinx development tools (Vivado and Vitis) as well as the Petalinux tool.

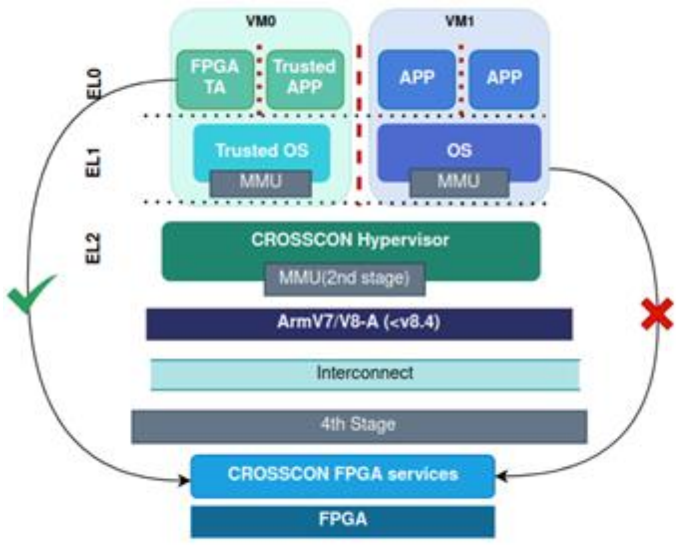

Figure 16: TA providing FPGA services on CROSSCON Hypervisor

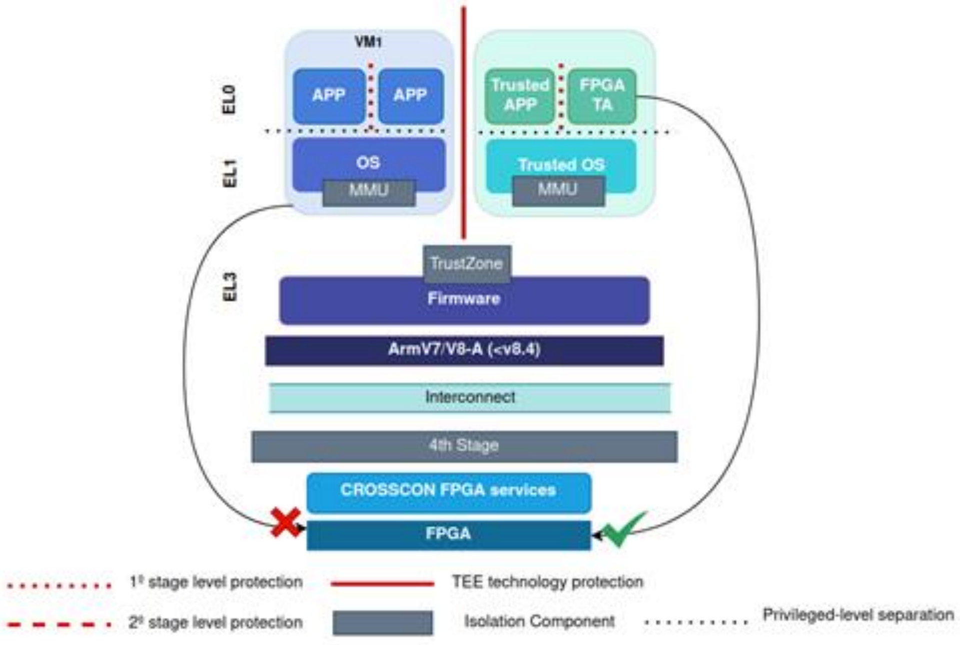

Figure 17: TA providing FPGA trusted services on TrustZone

8 Conclusion

In this document, we present the initial research and development results of work done as part of WP4. We have proposed the initial architecture of the CROSSCON SoC including the design of BA51-H (RISC-V) core, which can be used together with the CROSSCON Hypervisor to provide virtualization-based TEEs: a novel approach towards establishing TEEs on RISC-V architecture. We have provided a reference implementation of unified SPMP extension: a suggested approach for how to extend the RISC-V SPMP specification that was proposed as part of the CROSSCON project. Furthermore, we have suggested a solution based on the PG that allows sharing of domain specific HW modules across VMs while preserving isolation between them. We have considered how different HW modules can be used from SW perspective and researched possible interfaces that could be used to expose the functionality provided by the HW modules to CROSSCON trusted services and other applications. Furthermore, we have suggested the initial solution and design for how the FPGA fabric can be shared by multiple tenants on a TEE-like environment that allows spatial multi-tenancy while protecting IP designs before and after their deployments through strict security measures. The results described in this document are accompanied by demonstrators, code and bitstreams that are a part of D4.2 and are available on Github [51][52] [21].

This deliverable D4.1 contributes to the accomplishment of milestone MS4 "First version of the CROSSCON Stack components and extension primitives, and testbed" and provides the necessary insight for further work in WP3 and WP4. Furthermore, it provides the information necessary for the integration and validation effort done as part of the WP5.

As part of our future work aimed towards MS8 "Final version of the CROSSCON Stack components and extension primitives, and testbed", we plan to specify the interfaces for other domain-specific hardware components utilized by CROSSCON trusted services. We will refine the design of the CROSSCON SoC and implement a working prototype that addresses the changes in the design. We will continue our work on the PG by trying out other operation modes and how it can be integrated into the rest of the system. Further, we are planning to refine the design of the TA_{FPGA} and the FPGA shell into two separate entities, such that the TA_{FPGA} can run externally on top of an ARM core, and they allow to enforce access control to vFPGAs. As for the control-flow attestation extension, we are planning to implement the pre-processing accelerator and investigate feasible integration options on the ZCU102.

References

[1] ARM, "CryptoCell-300 Family," [Online]. Available: https://www.arm.com/products/silicon-ip-security/crypto-cell-300. [Accessed 19 March 2024].

[2] N. Semiconductor, "CryptoCell API," [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v15.0.0%2Fgroup__cryptocell__api.html. [Accessed 22 March 2024].

[3] NXP, "Hash Engine," [Online]. Available: https://www.nxp.com/docs/en/application-note/AN12834.pdf. [Accessed 22 March 2024].

[4] NXP, "RT600 HASH Engine," [Online]. Available: https://www.nxp.com/docs/en/application-note/AN12834.pdf. [Accessed 22 March 2024].

[5] NXP, "Asymmetric Cryptographic Accelerator CASPER," [Online]. Available: https://www.nxp.com/docs/en/application-note/AN12445.pdf. [Accessed 2 April 2024].

[6] R. Electronics, "Secure Crypto Engine Operational Modes – Application Note," [Online]. Available: https://www.renesas.com/us/en/document/apn/secure-crypto-engine-operational-modes-application-note. [Accessed 21 March 2024].

[7] F. Semiconductor, "ColdFire/ColdFire+ CAU and Kinteis mmCAU," [Online]. Available: https://www.nxp.com/docs/en/user-guide/CAUAPIUG.pdf. [Accessed 3 April 2024].

[8] Infineon, "Crypto (Cryptography)," [Online]. Available: https://infineon.github.io/psoc6pdl/pdl_api_reference_manual/html/group__group__crypto.html. [Accessed 22 March 2024].

[9] Infineon, "Cryptolite (Cryptography)," [Online]. Available: https://infineon.github.io/mtb-pdl-cat1/pdl_api_reference_manual/html/group__group__cryptolite.html. [Accessed 3 April 2024].

[10] B. e. a. Peccerillo, "A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives," Journal of Systems Architecture, 2022.

[11] L. P. e. a. Ledwaba, "Performance costs of software cryptography in securing new-generation Internet of energy endpoint devices," IEEE Access 6, 2018.

[12] ARM, "PSA Certified Crypto API," 2024. [Online]. Available: https://arm-software.github.io/psa-api/crypto/1.1/.

[13] GlobalPlatform, "GlobalPlatform TEE Internal Core API Specification v1.2," [Online]. Available: https://globalplatform.org/wp-content/uploads/2018/06/GPD_TEE_Internal_Core_API_Specification_v1.1.2.50_PublicReview.pdf.

[14] P. Ravikanth, B. Recht, J. Taylor and N. Gershenfeld, "Physical One-way functions," Science, pp. 2026-2030, 2002.

[15] Xilinx, "Github," [Online]. Available: https://github.com/Xilinx/embeddedsw/blob/master/lib/sw_services/xilskey/src/xilskey_eps_zynqmp_puf.c. [Accessed 3 April 2024].

[16]M. Integrated, "MAX32520 - ChipDNA Secure Arm Cortex M4 Microcontroller," [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/MAX32520.pdf. [Accessed 3 April 2024].

[17]PUFsecurrity, "Hardware root key generation and storage that never leaves the chip," [Online]. Available: https://www.pufsecurity.com/products/pufrt/. [Accessed 3 April 2024].

[18]PUFsecurity, "A PUF-based Crypto Coprocessor," [Online]. Available: https://www.pufsecurity.com/products/pufcc/. [Accessed 3 April 2024].

[19]Synopsis, "Synopsys Software PUF IP (Zign)," [Online]. Available: https://www.synopsys.com/designware-ip/security-ip/cryptography-ip/puf/software-puf.html. [Accessed 3 April 2024].

[20]CROSSCON, "API Demonstrator," [Online]. Available: https://github.com/crosscon/crypto_API_demonstrator.

[21]"CROSSCON SoC repository," April 2024. [Online]. Available: https://github.com/crosscon/crosscon_soc.

[22]"Beyond Semiconductor," [Online]. Available: https://www.beyondsemi.com/. [Accessed April 2024].

[23]"Spike RISC-V ISA Simulator," [Online]. Available: https://github.com/riscv-software-src/riscv-isa-sim. [Accessed April 2024].

[24]"Extended Spike RISC-V ISA Simulator," [Online]. Available: https://github.com/crosscon/riscv-isa-sim. [Accessed April 2024].

[25]"RISC-V SPMP Specification," RISC-V SPMP Task Group, [Online]. Available: https://github.com/riscv/riscv-spmp. [Accessed April 2024].

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

[27]ARM, "AMBA 3 AHB-Lite Protocol Specification (Version A)," 2006. [Online]. Available: https://developer.arm.com/documentation/ihi0033/a/?lang=en.

[28]CROSSCON WP2, "D2.1 CROSSCON Specification - Draft," 2023.

[29]J. Rushby, "Noninterference, Transitivity, and Channel-Control Security Policies," Computer Science Laboratory SRI International, 2005.

[30]CROSSCON WP3, "D3.1 CROSSCON Open Security Stack Documentation - Draft," 2024.

[31]D. Koch, Partial reconfiguration on FPGAs: architectures, tools and applications, Springer Science & Business Media, 2012.

[32]C. Plessl and M. Platzner, "Virtualization of hardware - introduction and survey," in International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), 2004.

[33]A. Vaishnav, K. D. Pham and D. Koch, "A survey on FPGA virtualization," in IEEE International Conference on Field Programmable Logic and Applications (FPL), 2018.

[34]S. A. Fahmy, K. Vipin and S. Shreejith, "Virtualized FPGA accelerators for efficient cloud computing," in IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2015.

[35]A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza and C. J. Rossbach, "Sharing, protection, and compatibility for reconfigurable fabric with amorphos," in USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2018.

[36]S. yma, J. G. Steffan, H. Bannazadeh, A. L. Garcia and P. Chow, "FPGAs in the cloud: booting virtualized hardware accelerators with openstack," in IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2014.

[37]O. Knodel, P. Lehmann and R. G. Spallek, "RC3E: reconfigurable accelerators in data centres and their provision by adapted service models," in IEEE International Conference on Cloud Computing (CLOUD), 2016.

[38]I. Giechaskiel, K. B. Rasmussen and K. Eguro, "Leaky wires: information leakage and covert communication between FPGA long wires," in Asia Conference on Computer and Communications Security (ASIACCS), 2018.

[39]I. Giechaskiel, K. Rasmussen and J. Szefer, "C3APSULe: cross-FPGA covert-channel attacks through power supply unit leakage," in IEEE Symposium on Security and Privacy (S&P), 2020.

[40]D. Mahmoud and M. Stojilovic, "Timing violation induced faults in multi-tenant FPGAs," in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2019.

[41]S. Zeitouni, G. Dessouky and A.-R. Sadeghi, "SoK: Secure FPGA Multi-Tenancy in the Cloud: Challenges and Opportunities.," in IEEE European Symposium on Security and Privacy (Euro S&P), 2021.

[42]F. Schellenberg, D. R. Gnad, A. Moradi and M. B. Tahoori, "An inside job: remote power analysis attacks on FPGAs," in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018.

[43]M. Zhao and E. G. Suh, "FPGA-based remote power side-channel attacks," in IEEE Symposium on Security and Privacy (S&P), 2018.

[44]F. Turan and I. Verbauwhede, "Trust in FPGA-accelerated cloud computing," in ACM Computing Surveys (CSUR), 2020.

[45]J. Vliegen, M. M. Rabbani, M. Conti and N. Mentens, "SACHa: Self-Attestation of Configurable Hardware," in Design, Automation \& Test in Europe Conference \& Exhibition (DATE), 2019.

[46]O. Glamocanin, L. Coulon, F. Regazzoni and M. Stojilovic, "Are cloud FPGAs really vulnerable to power analysis attacks?," in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2020.

[47]J. Krautter, D. R. Gnad and M. B. Tahoori, "Mitigating Electrical-level Attacks Towards Secure Multi-Tenant FPGAs in the Cloud," in ACM Transactions on Reconfigurable Technology and Systems (TRETS), 2021.

[48]T. La, K. Matas, N. Grunchevski, K. Pham and D. Koch, "FPGADefender: Malicious Self-Oscillator Scanning for Xilinx UltraScale+ FPGAs," in ACM Transactions on Reconfigurable Technology and Systems (TRETS), 2020.

[49]A. Xilinx, "Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit," [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

[50]A. Xilinx. [Online]. Available: https://docs.amd.com/r/en-US/xapp1353-pl-isolation.

[51]"API Demonstrator repository," [Online]. Available: https://github.com/crosscon/API_Demonstrator.

[52]"FPGA TEE repository," [Online]. Available: https://github.com/crosscon/FPGA_TEE. [Accessed April 2024].

[53]ARM, "AMBA 3 APB Protocol Specification v1.0," 2004. [Online]. Available: https://developer.arm.com/documentation/ihi0024/b/?lang=en.

[54]J. Rushby, "Noninterference, transitivity, and channel-control security policies," SRI International, Computer Science Laboratory, 1992.

[55]"RISC-V SPMP Extension," [Online]. Available: https://github.com/riscv/riscv-spmp.

Annex I: PG in time-sharing with reset mode specification

For the sake of completion, some of the information about the PG are repeated in this section. This section was prepared as an exercise in HW module design before the actual implementation to determine possible design and implementation problems. The section might be removed in future version of the document.

Perimeter guard is a HW module that can be used to control access to a slave module on a bus (e.g. HW accelerator) by placing it in front of the slave in such a way that all the communication between the masters and the slave goes through the PG. The PG can operate in different modes of operations, where in the following section we describe the time-sharing with reset and external arbitration mode.

For a more concise description of PG see chapter 6.

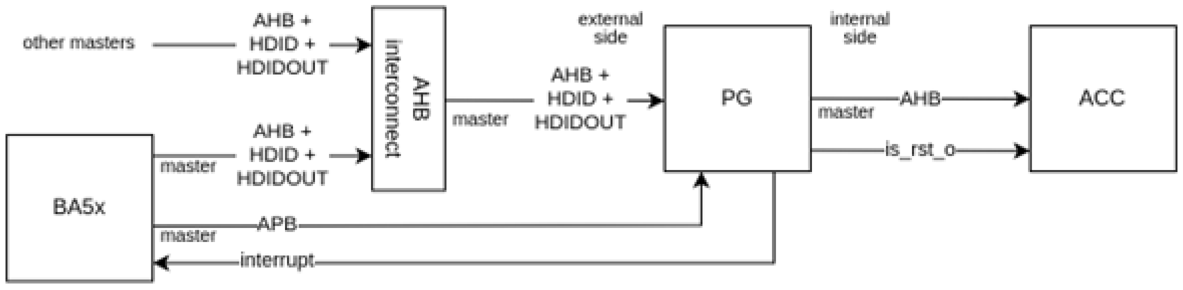

Figure 18: Overview how PG is connected to other HW modules.

Figure 18 shows how we connect a PG with other HW modules:

- The BA5x and other masters are connected to the PG with AHB over the interconnect. AHB has two additional signals, called HDID and HDIDOUT. HDID signal is driven by a master and HDIDOUT is driven by the PG. BA5x and other masters are the masters on the bus and the PG is the slave.
- The PG is connected to the ACC with AHB and a separate reset signal is_rst_o. The PG is the master and the ACC is the slave.
- BA5x is connects to the PG through a separate APB bus that is exclusively used to configure the PG. BA5x is the master and the PG is the slave.
- BA5x is connected to the PG with an interrupt signal that is driven by the PG.

For further discussion, we say that a PG has two sides: the side connected to interconnect, called *external side* (ES), and the side connected to the accelerator, called the *internal side* (IS). We often refer to sides when we need to clarify which signal has to be driven. Furthermore, we say that the master on the ES side is *ES master* and the slave on the IS side is the *IS slave*.

We assume that each transfer over a AHB is performed on behalf of a domain with a unique DID. Master transmits the DID alongside the transfer using HDID signals to let the slave know on behalf of which domain the transfer is performed. Note, for the first version of the PG, the BA5x and other masters are connected to the PG through AHB-Lite [27] bust that has two additional signals: HDID and HDIDOUT.

The configuration registers of the PG can only be configured through the APB bus (version B) [53]. We assume that all the APB masters are trusted. Usually, there will be only one APB master. We use RISC-V's Write Any Values, Reads Legal Values (WARL) abbreviation to describe behaviour of a register / field.

Time-sharing with reset mode: Functional description

In the *time-sharing with reset mode* a PG restricts access to the IS slave to a single domain at the time, and it resets the IS slave to a predefined state when the access to the IS slave is granted to another domain.

A PG restrict the access to the IS slave by *filtering* signals according to the set of rules: for example, only when AHB address and control signals are in a "valid" state and HDID == pg_did_reg, the signals are passed through from the ES master to the IS slave; otherwise, they are filtered out, i.e. instead of passing them through, the PG drives the signals to their default values on the IS.

Figure 19 shows the initial design of the PG. We divide the design into several submodules that handle specific parts of PG's behavior: (1) filters that are responsible for filtering different AHB signals (*AHB address and configuration signals filter (AHB ACS filter), AHB HWDATA filter* and *AHB response filter*); (2) *slave reset logic* responsible for resetting the slave, (3) *error response logic* responsible to signal an error over the interrupt or the AHB bus; and (4) *byd_ahb_pg_cfg* logic responsible for providing the configuration interface.

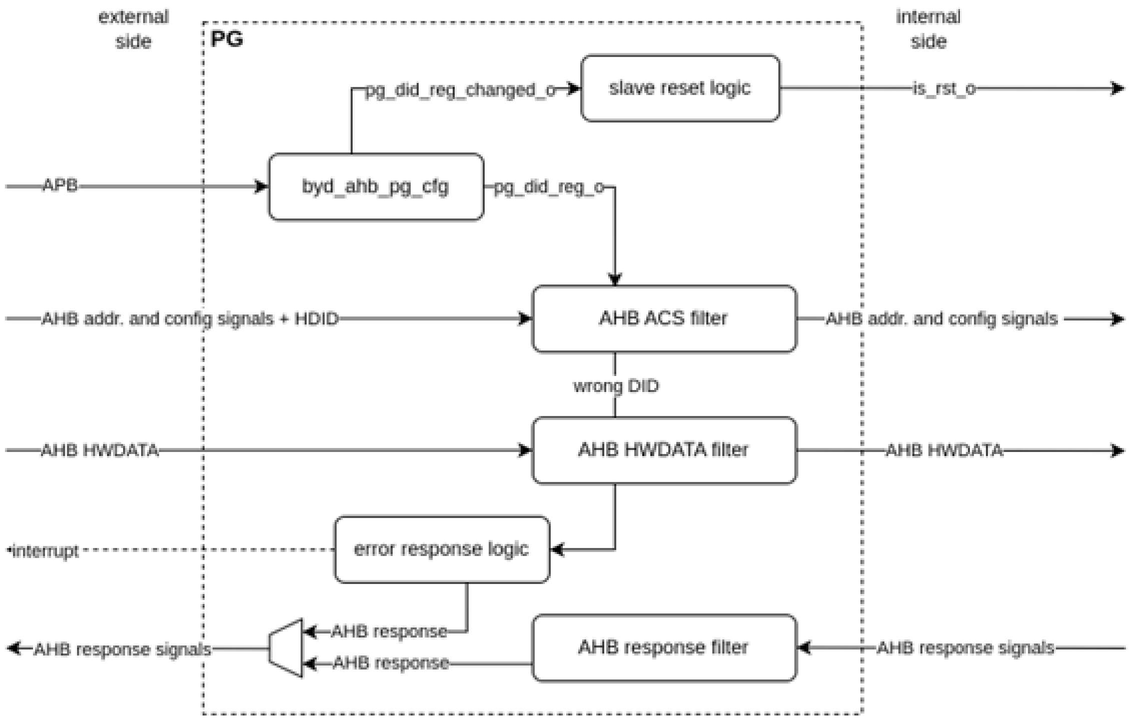

Figure 19: High-level overview of the design of a PG in time-sharing with reset mode

Note, to simplify Figure 19, not all the connections between different parts of the PG are drawn, e.g. all the filters are aware in which part of the error reporting or reset procedure the PG is in.

The PG is implemented as Verilog module with the signals listed in Table 4, Table 5, Table 6 and Table 7.

Table 4: PG's external side (ES) input signals

Name	Source	Size	Description
clk_i	global clock	1	
rst_i	global reset	1	
AHB address and control signals	ES AHB master		AHB's address and control signals driven by the master at the address phase. This includes:
- es_ahb_hsel_i	ES AHB master	1	AHB HSEL signal
- es_ahb_hready_i	ES AHB master	1	AHB HREADY signal
- es_ahb_haddr_i	ES AHB master	AHB_AW	AHB HADDR signal
- es_ahb_htrans_i	ES AHB master	2	AHB HTRANS signal

- es_ahb_hwrite_i	ES AHB master	1	AHB HWRITE signal
- es_ahb_hsize_i	ES AHB master	3	AHB HSIZE signal
- es_ahb_hburst_i	ES AHB master	3	AHB HBURST signal
- es_ahb_hprot_i	ES AHB master	3	AHB HPROT signal
- es_ahb_hmastlock_i	ES AHB master	1	AHB HMASTLOCK signal
AHB HDID signal	ES AHB master		
- es_ahb_hdid_i	ES AHB master	32	The ID of the domain that caused the transfer. Driven by the master in the address phase of a transfer and has the same timing as the AHB HADDR.
AHB HWDATA signal	ES AHB master		
- es_ahb_hwdata_i	ES AHB master	AHB_DW	AHB HWDATA signal
APB signals	ES APB master		Part of the APB interface through which the PG can be configured. This includes:
- es_apb_psel_i	ES APB master	1	APB PSEL signal
- es_apb_paddr_i	ES APB master	32	APB PHADDR signal
- es_apb_pwrite_i	ES APB master	1	APB PWRITE signal
- es_apb_penable_i	ES APB master	1	APB PENABLE signal
- es_apb_pwdata_i	ES APB master	32	APB PWDATA signal

Table 5: PG's external side (ES) output signals

Name	Destination	Size	Description
AHB response signals	ES AHB master		
- es_ahb_hreadyout_o	ES AHB master	1	AHB RESP signal
- es_ahb_hresp_o	ES AHB master	1	AHB HRESP signal
- es_ahb_hrdata_o	ES AHB master	AHB_DW	AHB HRDATA signal
HDIDOUT			
- es_ahb_hdidout_o	ES AHB master	32	Driven by the PG. Needs to always be in a valid state; except during reset. Has the same value as the pg_reg_did register. HDIDOUT represents a domain ID of the domain that can currently access the ACC. This signal is not relevant for the basic setup of the PG. But when we also have PG_out, this signal allows us to enforce additional security guarantees for the entire interconnect.
APB signals	ES APB master		Part of the APB interface through which the PG can be configured. This includes:
- es_apb_pready_o	ES APB master	1	APB PREADY signal
- es_apb_prdata_o	ES APB master	32	APB PRDATA signal
- es_apb_pslverr_o	ES APB master	1	APB PSLERR signal
es_interrupt_o	a core responsible for handling interrupts (e.g. BA5x); usually the same	1	An interrupt signal indicating when an error has occurred.

	core that can access PG's CSRs over APB		

Table 6: PG's internal side (IS) input signals

Name	Source	Size	Description
AHB response	IS slave		
- is_ahb_hreadyout_i	IS AHB slave	1	AHB HREADYOUT signal
- is_ahb_hresp_i	IS AHB slave	1	AHB HRESP signal
- is_ahb_hrdata_i	IS AHB slave	AHB_DW	AHB HRDATA signal

Table 7: PG's internal side (IS) output signals

Name	Destination	Size	Description
AHB address and control signals	ACC		
- is_ahb_hsel_o	IS slave	1	AHB HSEL signal
- is_ahb_hready_o	IS slave	1	AHB HREADY signal
- is_ahb_haddr_o	IS slave	AHB_AW	AHB HADDR signal
- is_ahb_htrans_o	IS slave	2	AHB HTRANS signal
- is_ahb_hwrite_o	IS slave	1	AHB HWRITE signal
- is_ahb_hsize_o	IS slave	3	AHB HSIZE signal
- is_ahb_hburst_o	IS slave	3	AHB HBURST signal
- is_ahb_hprot_o	IS slave	3	AHB HPROT signal
- is_ahb_hmastlock_o	IS slave	1	AHB HMASTLOCK signal
AHB HWDATA signal	IS slave		
- is_ahb_hwdata_o	IS slave	AHB_DW	AHB's HWDATA signal
is_rst_o	IS slave	1	When driven HIGH, the IS slave should be reset to predefined state.

The PG has the following configuration registers pg_did_reg, pg_cfg_reg and pg_interrupt_cause_reg that can be accessed through the APB interface as described in description of byd_ahb_pg_cfg module.

When the PG wants to reset the IS slave, it will drive is_rst_o wire HIGH. When is_rst_o wire is driven HIGH, the IS slave needs to be reset to a predefined state. Note that the reset procedure performed by the IS slave might take several cycles to complete. The number of cycles needed by the IS slave to reset can be configured through the S_RST_DELAY parameter of a byd_ahb_pg module. The PG drives the is_rst_o HIGH for the duration of the reset and drives the es_ahb_hready_o signal LOW to create a back pressure on ES.

Workflow:

1. Upon reset:
    - pg_did_reg == DEF_DID where DEF_DID parameter can be configured at integration.
2. The APB master can grant access to the IS slave to another domain by writing its ID to the pg_did_reg register.
    - The pg_did_reg register can be accessed through the APB interface. See the byd_ahb_pg_cfg description.
    - When pg_did_reg is written to, the IS slave is reset. See the slaves reset logic description.

3. When a valid address or data phase signals of a transfer are received by the PG over AHB, the PG determines if the signals can be passed to / from the IS slave according to the set of rules as described in AHB ACS filter, AHB HWDATA filter and AHB response filter descriptions:
    - If signals of the address phase meet the rules (e.g. pg_did_reg == es_ahb_hdid_i), they are passed through to the IS slave; otherwise, an error is raised and the corresponding output signals on the IS are driven to default values. See the error response logic description.
    - If signals of the data phase meet the rules, they are passed through to / from the IS slave; otherwise, the corresponding output signals on the IS / ES are driven to default values.

Description of byd_ahb_pg_cfg module:

The byd_ahb_pg_cfg module contains the PG's configuration registers (Table 8) and allows them to be updated through the APB interface.

Table 8: PG's configuration registers

Register	Size (in bits)	Address[8:2] (in hex)	Description
pg_did_reg	32	00	(WARL) A register containing the ID of a domain that can access the IS slave.
pg_cfg_reg	32	01	(WARL) A configuration registers where:
pg_cfg_reg[0]	1		- report_error_over_bus_bit - When set to 1, an error is signalled over AHB bust by raising the es_ahb_hresp_o signal.
pg_cfg_reg[1]	1		- report_error_over_interrupt_bit - When set to 1, an error is signalled over the interrupt wire.
pg_cfg_reg[2:31]	30		- Bits from 2 to 31 are reserved.
pg_interrupt_cause_reg	32	02	(Read only) A register that contains the DID of the last domain that caused an interrupt/error.

All configuration registers are big-endian.

When the pg_did_reg register is written to, the IS slave is reset as described in slave reset logic.

Assumptions: We assume that all the masters connected to the APB interface are trusted.

Description of AHB ACS filter:

The AHB ACS filter filters the AHB address and control signals according to the following rules:

- address and control signals are valid, i.e. es_ahb_hsel_i == HIGH and es_ahb_hready_i == HIGH,
- es_ahb_htrans_i != IDLE,
- the PG is not in the 1st cycle of reporting an error,
- the IS slave is not being reset and
- es_ahb_hdid_i == pg_did_reg.

If the rules are met, the signals are passed from the ES master to the IS slave; otherwise, an error is raised, and PG drives the signals to their default values on the IS. See error response logic functional description for how error is reported.

Description of AHB HWDATA filter:

The AHB HWDATA filer filters the HWDATA signal according to the following rules:

- there is a transfer in an address phase, i.e. time between a valid address phase and until the IS slave returns a response,
- the PG is not reporting an error and
- the IS slave is not being reset.

If the rules are met, the HWDATA signal is passed from the ES master to the IS slave; otherwise, PG drives is_ahb_hwdata_o signal to its default value on the ES.

Description of AHB response filter:

The AHB response filer filters the IS slave's response signals (is_ahb_hready_i, is_ahb_hresp_i and is_ahb_hrdata_i) according to the following rules:

- there is a transfer in an address phase, i.e. time between a valid address phase and until the IS slave returns a response,
- the PG is not reporting an error and
- the IS slave is not being reset.

If the rules are met, the signals are passed through; otherwise, PG drives the signals to their default values on the ES.

Description of slave reset logic:

When the slave reset is triggered (e.g. when pg_did_reg is written to), slave reset logic resets the IS slave by driving the is_rst_o signals HIGH for S_RST_DELAY number of clock cycles.

S_RST_DELAY parameter should be set at integration time to the number of cycles needed by IS slave to be reset. Default value is 1 cycle.

Note, we call the period, when the is_rst_o is driven HIGH to reset the IS slave, the *IS slave's reset cycle* or just *reset cycle* when it is clear from the context to which reset cycle we refer to.

During the reset cycle, all the incoming signals from ES and IS side on AHB bus are ignored and the PG drives the output signals to their default values. Note that during reset the ES es_ahb_hreadyout_o is driven LOW, which delays any transfers send to the IS slave which crates back pressure on the bus.

If a slave reset is triggered when the PG is already in the *reset cycle*, the *reset cycle* is repeated.

Description of error response logic:

When an error occurs during a transfer, the PG can signal an error to the ES master initiating the transfer over the AHB bus and/or to the BA5x core as an interrupt over es_interrupt_o signal. An error is only signalled over the AHB bus or as interrupt if report_error_over_bus_bit or report_error_over_interrupt_bit of pg_cfg_reg are set to 1 respectively; otherwise, the error is not reported. An error is reported as a response to a transfer over the AHB bus as specified by the AHB-Lite [27] specification in chapter 5.1.3 ERROR response. An error is reported over es_interrupt_o by driving the signal HIGH for one cycle when error occurs; otherwise, the interrupt signal is driven LOW.

A PG raises an error when an ES AHB master starts a transfer but it currently does not have access to the IS slave (i.e. es_ahb_hdid_i != pg_did_reg); more precisely, when the following conditions hold:

- address and control signals are valid, i.e. es_ahb_hsel_i == HIGH and es_ahb_hready_i == HIGH',
- es_ahb_htrans_i != IDLE,
- the PG is not in the 1st cycle of reporting an error, i.e. when PG drives es_ahb_hready_i == LOW and es_ahb_hresp_i == HIGH,
- the IS slave is not being reset and
- es_ahb_hdid_i != pg_did_reg.

When an error occurs, an ES APB master can determine which domain has caused the error by reading the DID from the pg_interrupt_cause_reg register over the APB bus.

Note that the es_ahb_hdid_i signal should remain the same for the duration of a burst as expected for most of other AHB control and address signals. From this it follows that, the only time when the PG could raise an error during a burst is if the PG's pg_did_reg changes during a burst which can only happen if the ES APB master reconfigures the PG.