POLITECNICO DI TORINO

Master's Degree in CYBERSECURITY

Master's Degree Thesis

DESIGN AND EVALUATION OF HARDWARE SECURITY MODULE ON RISC-V FOR AUTOMOTIVE
A SECURE BOOT CASE STUDY

Supervisors

Prof. ALESSANDRO SAVINO

Prof. STEFANO DI CARLO

Dr. FRANCO OBERTI

Candidate

OZGE FEDAI

MARCH 2026

Abstract

Automotive architecture has become a large network of Electronic Control Units (ECUs). From critical ECUs that can affect passengers' lives to less critical units coexist in the network. The complexity of the system is intensified over the external data sources including sensors or Over-The-Air (OTA) updates. Increasing electrification and software-defined vehicles made OTA updates a frequent and necessary part of the product lifecycle to fix vulnerabilities. But threat actors can inject malicious firmware during these updates, bypass security checks, or perform unauthorized access to the car's internal network. Hardware Security Modules (HSMs) serve high-performance secure cryptography computations for real-time applications in an isolated environment. Mitigates risks arising from vulnerabilities. HSM can realize a hardware root of trust, which is the trust anchor of the entire system. Certified proprietary HSMs exist, but their closed architectural designs are not extensible due to fixed silicon designs. Their cryptographic capabilities are restricted to vendor-defined instruction sets, preventing independent security verifications of HSM's trustworthiness. Consequently, these limitations prevent the platform from being extended or integrated with new security features such as post-quantum algorithms. In the long term, this could lead to a weak security posture for future threat models. Research becomes tied to a specific platform, rather than transferable design blueprints across different hardware architectures. This work uses Parallel Ultra Low Power (PULP) platform to align with modern automotive ECU's an open-source RISC-V instruction set architecture (ISA) that is extendable. Explores the feasibility of architectural design and implementation of an HSM component that can functionally demonstrate secure boot early stages. In a GVSoC simulation environment, two design strategies were explored for HSM architectural isolation. (i) MMIO peripheral model of HSM (ii) Minimal ISA extension with custom HSM instruction. The results showed a practical limitation of the MMIO in our chosen core model, due to the lack of Physical Memory Protection (PMP). Missing PMP enforcement leads to exposure of key material to untrusted software. The demonstration of HSM performing secure boot, therefore, completely focuses on the strategy (ii) to realize boot image integrity and authenticity.
Case study of secure boot implements Boot ROM verification of the primary boot loader (PBL). Verification stage abstracts asymmetric cryptography to simple keyed digest to effectively model side channel leak. Secret key in keyed digest is only accessible by the custom instruction handler and is not present in a shared memory space. Analysis of attacker models and threads showed that Denial of Service (DoS) is possible in the absence of protection by the invocation of a custom HSM operation. Moreover, GVSoC's cycle accuracy feature is used to firsts realistically

modeling time for computational costs then do demonstrate temporal side-channel vulnerability.
Time leak side channel may break the symmetric cryptography through byte-by-byte key compromise; asymmetric cryptography doesn't show this phenomenon due to the underlying mathematical operations and no secrecy of public material that is stored in HSM. This study proposed a transparent design and extensible framework for HSM architectural isolation. It allows the microarchitecture-level threat model and attack path analysis as open security for future works.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my family for their unshakable belief in me and for the unconditional support they have provided throughout my life.

I also owe my sincere and heartfelt thanks to my supervisor, Professor Alessandro Savino, whose positive influence on both my design and implementation decisions throughout the making of this thesis has been truly invaluable. His guidance helped me find my own path through a subject of considerable depth and complexity.

I would also like to take this opportunity to express my gratitude to a few other individuals to whom I am deeply grateful:

I would like to thank my manager, Antonio Parrotta, whose guidance and mentorship since the beginning of my career greatly strengthened my interest and curiosity in the automotive field.

Finally, to my friends, thank you for the genuine happiness you expressed for my achievements and for the encouragement and warmth you brought to my life.

Table of Contents

List of Tables

List of Figures

Acronyms

CPA
Correlation Power Analysis

CPU
Central Processing Unit

CSR
Control Status Register

DPA
Differential Power Analysis

EMA
Electromagnetic Analysis

GVSoC
Pulp Chips Simulator

HSM
Hardware Security Module

ISA
Instruction Set Architecture

ISS
Instruction Set Simulation

MCU
Microcontroller Unit

MMIO

Memory Mapped Input Output

OEM

Original Equipment Manufacturer

OTA

Over-the-Air update

PK

Public Key

PMP

Physical Memory Protection

PULP

Parallel Ultra Low Power

RISC-V

Reduced Instruction Set Computing Fifth Version

RV32

32-bit RISC-V architecture

SoC

System on Chip

SK

Secret Key

Chapter 1

Introduction

1.1 Motivation

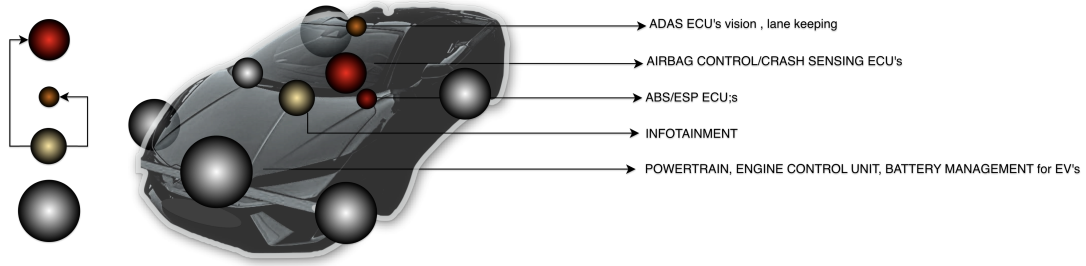

Figure 1.1: Vehicle's various ECU control systems based on importance illustrated

A modern automotive can include in the order of tens to over hundred Electrical Control Units (ECU's). Together, they make the underlying electronic mechanism of an automotive. The mechanism can be seen to be working in multiple different layers [1]; from hardware, firmware, operating system and application layers. Each responsible for distinct system functionalities.

While having this complex internal computing system network within a car as illustrated in Figure 1.1, from the outside, a modern car is also interacting with other vehicles or infrastructures, as seen in the Figure 1.2. Using Vehicle to Vehicle V2V and Vehicle to Everything (V2X),

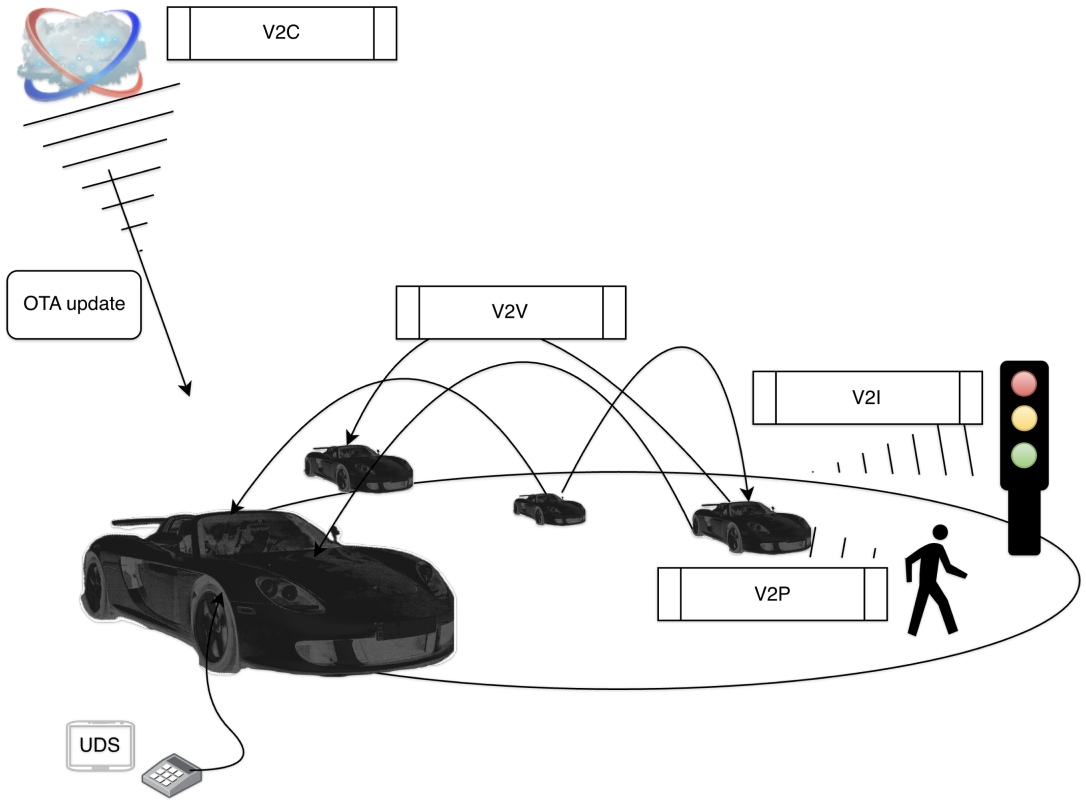

Figure 1.2: Automotive interaction with outside world

getting over-the-air updates conveniently with Over the Air Update (OTA) technologies. Only with an increasing pattern toward the future.

Both within and outside, the services i.e., Anti-lock braking system (ABS), airbag control a car relies on is provided and controlled by its ECU's. Some of which are highly critical and can impact the passenger's safety [2].

Cybersecurity is an essential part of the vehicle's security; it is tightly coupled with vehicles safety-functionality and is implemented to provide protection mechanisms against deliberate attacks or critical system failures.

Figure 1.1 illustrates the various ECU components and their role. It is a simplified demonstration, as it doesn't show dense internal network traffic over the communication channel. It shows real ECUs and their criticality with color codes. Damage to these ECUs can directly affect the *assets*, which can be defined as the value desired to be protected. In this context, the assets are passenger's life, the car itself and its financial value or even intellectual property of the vehicle components. Although the network part is abstracted, the diagram is intended

to capture the risk associated to this ECUs with color codes. Additionally, it captures the functional diversity of ECUs involved in a modern car that share the same space and affect each other. Such an interconnected and diverse computing system inherently possesses a large attack surface. An *Attack surface* concerns the interfaces and possibilities an attacker can utilize to gain unauthorized access to the system and use this to influence system behavior for a malicious intention. The critical interfaces do not need to be targeted directly; in fact, an indirect access is a known issue. As an example, low-critical infotainment system is used to reach high critical parts with lateral movements. A movement from external to internal critical parts. Therefore, the goal is to minimize the impact of the attack surface caused by a malfunctioning or an infected component due to deliberate attack by enforcing proper security mechanisms.

Among the high numbers and variety of protection practices, ***verification of the boot stages of an automotive ECU*** takes an important place.

The growing reliance on software (SW) updates as a consequence of growing shift towards Software Defined Vehicles (SDV), makes OTA secure update mechanism a growing necessity. Secure boot defines a trusted chain of *verification process* while the Hardware Root of Trust (HRoT) is the physical anchor that enforces trust to the processes. This is done in order to assure the software on already on the device is trusted and handles new software delivery securely.

HRoT can be described as a form of a minimal set of immutable hardware and Read Only Memory (ROM) code, that the system trust at the power on.

Consequently, for security and performance in computation speed, there is a need for a component that performs secure execution, protects the secrets (i.e., keys sensitive material), and starts the root of trust process. The component that is specialized for security and can perform this operations in practice is the Hardware Security Module (HSM).

1.2 Problem statement

While there are existing, state of the art proprietary HSM solutions which are certified, the underlying design and how the security is achieved is a black-box concept. Moreover, due to this closed design it is difficult to study trust anchors at instruction/microarchitectural levels.

As more SW functionality is introduced to vehicles and rapid increase of attack surface makes the vendor dependency a concern. In this thesis the focus is to leverage the open platform in a simulation environment to study secure boot architectural flow. Before the real silicon implementations of the SoC, it is necessary to perform functional analysis and more important for this work the threat modeling of the potential attack vectors.

1.2.1 Study Objectives and Open-Source Rationale

Most open-source, micro controller platforms do not provide an integrated HSM and in automotive systems in particular existing HSM implementations are typically closed, vendor specific and not extensible by the research community. As a result, researchers cannot integrate custom cryptographic algorithms, perform experiments with mathematical constructions or modify security features beyond what is exposed by proprietary interfaces. This limitations significantly constrains the ability of cybersecurity research community to progress evaluate and expand HSM related knowledge. Especially in the context of future attack vectors and necessity for validation of post-quantum algorithms or domain specific cryptographic primitives.

In the state of the art, production-grade automotive HSM stacks such as Infineon [3] or NXP [4], which incorporate physically isolated core and vendor provisioned HRoT, represent the current industrial reference point for the automotive security. While these solutions achieve high effectiveness in production and certifications, the fixed HRoT implementations and closed micro-architectural designs significantly limit the ability of researchers to modify, instrument and perform independent security verification experiments. Moreover, as the future attack vectors requires repeatability in and adaptation of the security algorithms thus verification of security critical components also need continuity. The intent of this work is neither to replace certified automotive HSM's, nor to claim compliance with industrial certification standards. Rather, it aims to provide research, oriented, open and extensible HSM framework that enables machine level emulation of an open-source platform execution. Including security practices, verification strategies and controlled cryptographic operations. Allowing experimentation with secure boot and opcode level trust primitives. This level of control and knowledge of a system is inaccessible in most of the proprietary HSM implementations.

1.3 Contributions

This work contributes to the research in this field by implementing an open-source design of an HSM primitive component and its creation inside the SoC using existing PULP cores [5] with the ISA-offloading strategy utilizing custom instructions and ISA-extension. The CPU pipeline is simulated with GVSoC [6], a cycle accurate simulator for PULP chips that supports SW development, timing and power analysis. The initial strategy was to model the HSM as separate isolated component however due to the experimental results the ISA extension strategy is chosen. GVSoC can simulate event *timing* effectively including pipeline stalls, memory delays producing traceable cycle times.

Objectives to achieve throughout this study is:

1. Secure key handling inside untrusted SW environment in a fully open-source SoC.

2. Structured architectural study of secure boot root of trust.

3. Analysis of security posture for 2 isolation strategies.
   (i) HSM as Memory Mapped Input Output (MMIO) with PMP protection.
   (ii) HSM as instruction level dedicated secure boot primitive.

4. Threat model evaluation for secure boot early stage of verification chain.

5. Isolation without modifying RTL, analysis of limitation in GVSoC simulation environment.

1.4 Thesis Organization

The following sections of this thesis is organized as follows.

Chapter 2 introduces the theoretical background with the definition of assets in automotive cybersecurity, the concept of secure boot and its relation to trust boundaries, automotive threats from a high and low-level perspective, the distinction of integrity property for data at transit and data at rest, what can happen to the firmware in a storage environment and a detailed process flow of secure boot in practice.

Chapter 3 presents the experimental environment of this thesis, covering the specific core choice within the PULP platform and the GVSoC simulator specification and its important features for this study.

Chapter 4 presents the two methodologies. It introduces the first design strategy to model HSM as an MMIO peripheral, and explores the rationale for verifying physical memory protection. Following this, it introduces the second design strategy, modeling secure boot verification as a custom ISA extension. This includes the early phases of the secure boot chain, immutable read-only memory enforcement, memory region separation and custom opcode signature verification. Implementation details of simple extended instruction, up to the timing modeling of signature verification is covered. The chapter concludes with the achieved reduced attack-surface.

Chapter 5, formalizes the threat model by defining specific attacker types, their capabilities and their attack visibility within the simulation environment.

It visually demonstrates attack path diagrams for both methodologies, and timing base considerations.

Chapter 6, presents the results of the experiments performed for the two methodologies. It makes a competitive evaluation of security posture for both HSM as a protected peripheral model and the ISA extension. Moreover it explains the

observations of the two threat modeling attack scenarios: time side channel analysis and fault injection modeling, and attack results. The chapter concludes with an evaluation of HSM over a simulation environment with respect to the real-world applications. It uses a summary table to consolidate the comparisons of different design strategies and their security implications.

Chapter 7, outlines the directions for the future research that builds on the open HSM designs.

Chapter 8, concludes the thesis by summarizing the architectural insights and achieved security properties, reflecting on isolation limitations inherent to the simulation based environment, and highlighting the open and modifiable nature of the platform as a foundation for the future HSM research.

is the conclusion that summarizes the architectural insights, achieved security properties. It presents the isolation limitations due to the simulation-based environment. open and modifiable platform for HSM implementation and direction towards the future research.

Chapter 2

Hardware Security Modules

The Hardware Security Module occupies a uniquely critical position in the System on Chip (SoC) security architecture. It serves as the system's trust anchor, the component from which all downstream security guarantees are derived. In the automotive context, this means that the integrity of the secure boot chain, the confidentiality of cryptographic keys, and the authenticity of software updates all converge on the HSM as the root of trust. Thus, this central role is precise of what makes the HSM the highest value target for an adversary. A successful attack against it does not compromise an individual feature, but it collapses the security posture of the entire system regardless of how well other components are hardened.

To fulfill this role, an HSM is expected to provide a rich set of security services. At its core, it maintains a dedicated secure key vault, which is a protected storage domain from which cryptographic material cannot be extracted by software operating outside the HSM's trust boundary. It is responsible for True Random Number Generation (TRNG) operation to ensure the unpredictability of generated keys and nonce. In addition, they support secure key provisioning, workflows, and provide long-term key storage with life-cycle management. Beyond key management, HSM functions as high-speed cryptographic accelerator offloading computationally intensive operations from the main CPU to itself. An HSM can perform symmetric encryption, cryptographic hash computation and asymmetric signature verification. This performance dimension and capabilities distinguish HSMs from Trusted Platform Module (TPM), where TPM's are designed for measured low throughput security operations. An HSM instead is made for high throughput cryptographic executions under the real-time constraints. Which is a requirement that is particularly stringent in automotive systems due to the latency concerns. Since HSM is one of the most complex components to prototype, analyze and reason about, particularly when the goal is to understand and validate the system's trust boundaries within the SoC design operating in a simulation environment. A complete HSM realization is therefore beyond the scope of this single study at

this stage. Instead, this thesis selects the foundational and security the critical subset of HSM functionality as its case study the secure boot operation. Figure 2.1 illustrates the first stage of the secure boot trust chain in a real-world deployment scenario. It is a reference point from which this work's architectural investigation of HSM design and threat modeling is grounded.

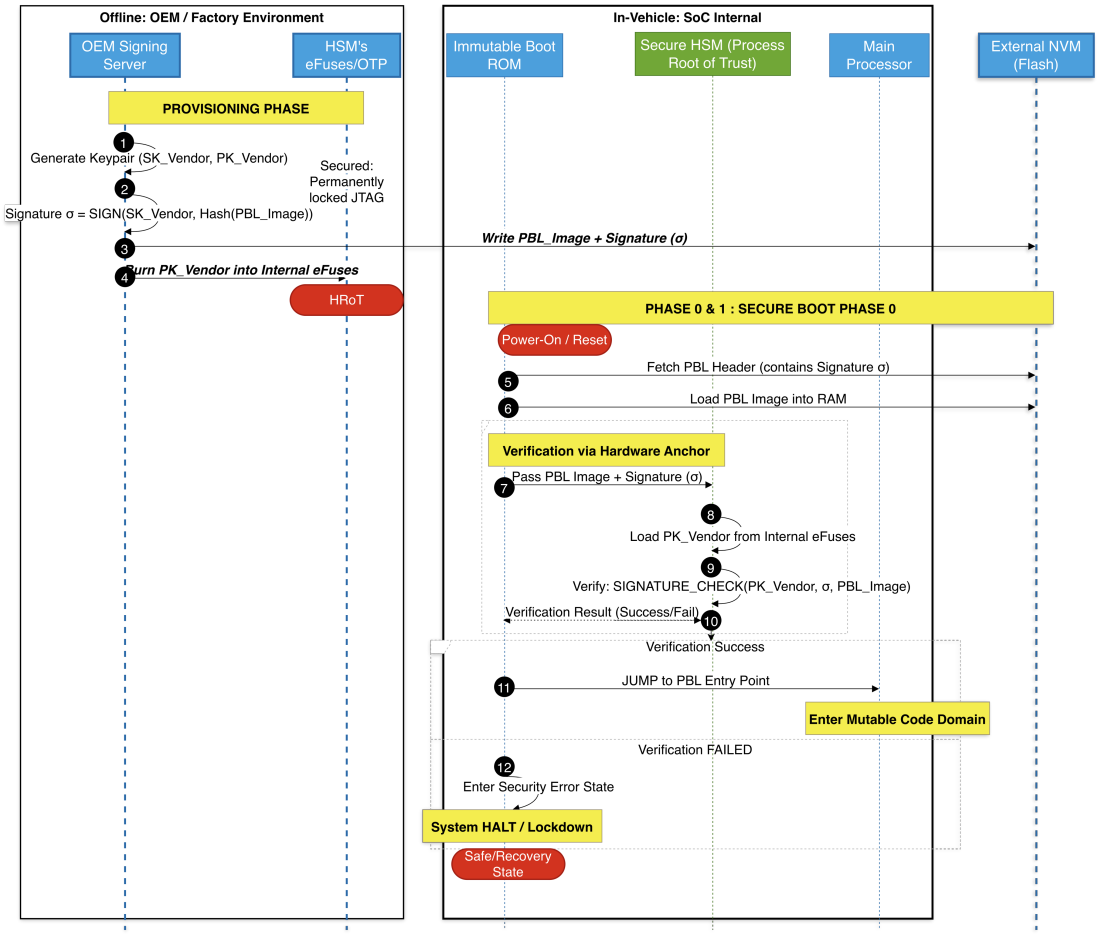

Figure 2.1: Boot ROM authenticates PBL (Primary Boot Loader)

2.0.1 HSM Case Study: Protecting the Integrity with Secure Boot

The case study for the HSM functionality to be demonstrated in this study is secure boot Phase 0, as shown in procedural details with figure 2.1. Among the scenarios HSM use i.e., random number generation, encryption, and decryption, the secure boot is chosen to be the foundation for the automotive industry, since in every boot

the ECU must use a public key signature and performs Primary Boot Loader (PBL) verification of Secondary Boot Loader (SBL). After this verification application is verified by the SBL. The software updates that are mentioned in the introduction relies on the security of this lower-level firmware verification mechanism. Since in physical silicon it relies on the HRoT, the tools to validate the key, memory and process management becomes foundational step in HSM implementation.

2.0.2 Assets

The security objectives are first defined by identifying the relevant assets considering both high-level systems and low-level implementation perspectives. Since the protection of system-level assets fundamentally depends on the integrity and trustworthiness of underlying hardware and software components. At the highest level, the ultimate goal for security is to provide passenger safety against injury risks. In addition, the protection of the vehicle against unauthorized access, hijacking, or malicious manipulation. This also concerns the financial damages. Achieving these goals requires ensuring the security of the underlying electronic control unit, particularly during software update procedures and inter-ECU communication. Where compromised integrity or authenticity of a low-level component can directly propagate to system-wide safety risks. As a consequence, cybersecurity mechanisms must be enforced at the execution levels to establish the trusted foundation for higher-level automotive functions.

Threat origin at high level to low-level

The Figure 2.2 is an illustration of high and low level threat scenarios and it is inspired by [7]. At the high level, threat are categorized by communication range: long range attack vectors such as remote adversaries that. Additionally the short range vectors including Bluetooth. Both cases represent entry points entry points through which a malicious data can be injected or tampered before reaching to the vehicle. At low level, once an adversary has gained access to the internal vehicle network, the CAN bus becomes the primary attack surface. At this point, a malicious Unified Diagnostic Service (UDS) can be used to flash unauthorized firmware.

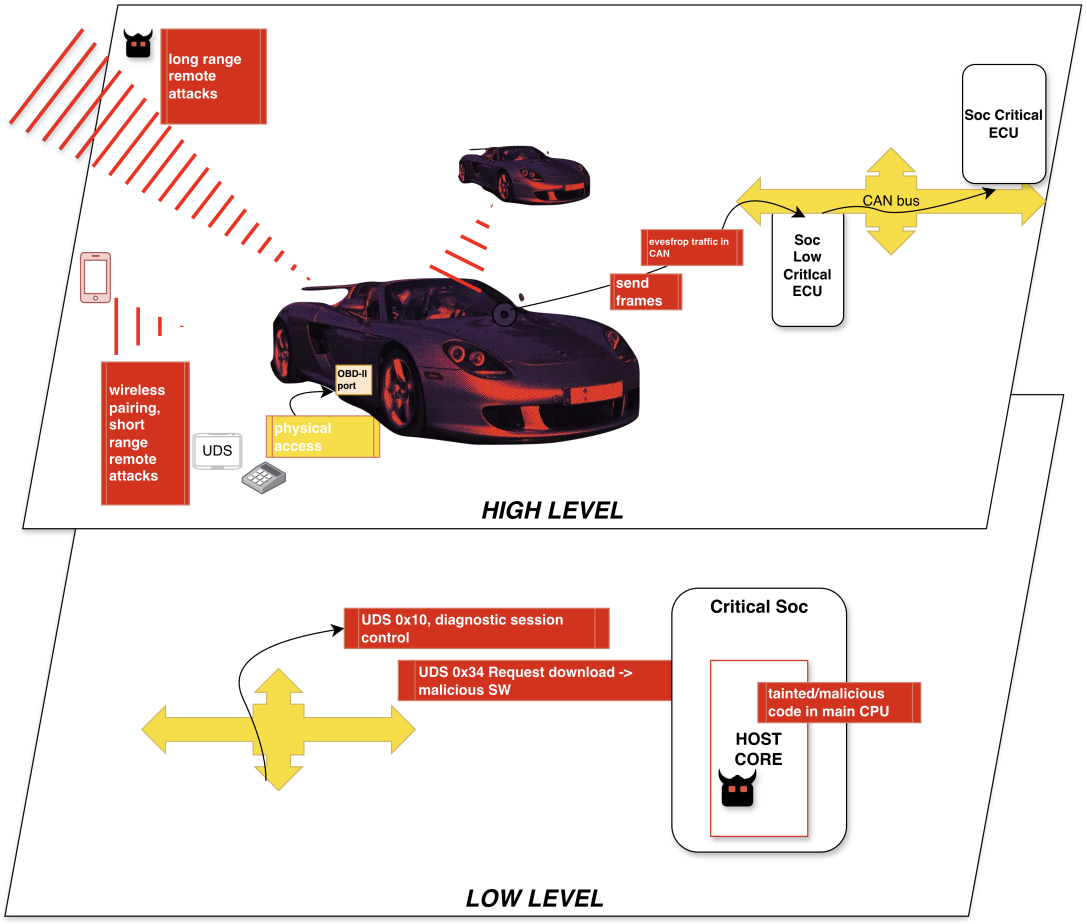

Figure 2.2: Illustration of threat models in high level and low level

2.0.3 Secure Boot Helps Framing the Trust Boundaries

The *chain of trust* in secure boot is one of the most essential flow in a security verification for the vehicle. It is a foundational process that forces security developers to model the core security concepts. It forces to ask where the trust anchor is and if the firmware image can be trusted. It is this process that assures trust over the entire system using HRoT.

Secure boot ensures that the system only begins execution after the immutable ROM code, that is the handled by the PBL. After PBL completes integrity check, the following this, each subsequent bootloader stage is verified by the preceding one. Essentially forming a chain of trust that extends from the ROM up to the final application software [8] pp.6. This is a widely adopted strategy in automotive ECU's. Figure 2.3 depict an example of cryptographic verification before allowing

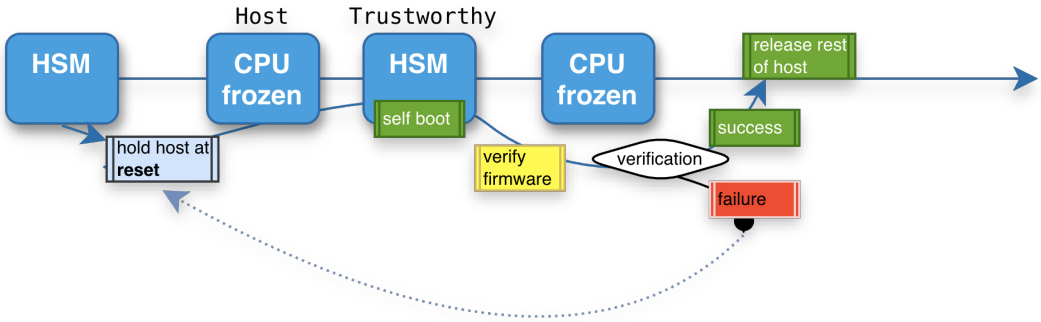

Figure 2.3: Host verified by HSM

the Host CPU to start. If the verification fails, HSM holds the CPU in a reset state.

2.0.4 Integrity property for data at rest or data in transit

Integrity can be explained with different possible scenarios. Among them, the common phenomenon is a change, and the detection of this change matters to preserve the real value or a state. In terms of data integrity protection, the requirement is that the data has not been modified. But this alone is not the only aspect. Another issue can be the cancellation of some data or the selection of only desired data in a transit channel that is going to arrive at the destination.

Cybersecurity protection can be achieved by means of protecting and completely avoiding possible attacks. As an example, the data secrecy concerns the protection of confidentiality, but the integrity protection is about the detection of the change; it doesn't claim to protect against the change, but it is about knowing if there has been any alteration. Realizing a situation awareness about the data status also necessitates the detection time being as early as possible. By demonstrating experiments this work presents the approach that will be covering one of the most important strategies in verifying the system integrity through a chain of verification which originates from a Root of Trust. In that regard, this scenario is specifically about the integrity of *data at rest*. Figure 2.4 shows the difference for the integrity property and the difference with respect to the data and its protection in transit, which is not concerned in this study. In the automotive context, during booting, it is a property to be assured that verifies the manufacturer's firmware is legitimate.

At this point, another security property to protect is observable: Data origin authentication. The creator of the data needs to be checked since it is the source

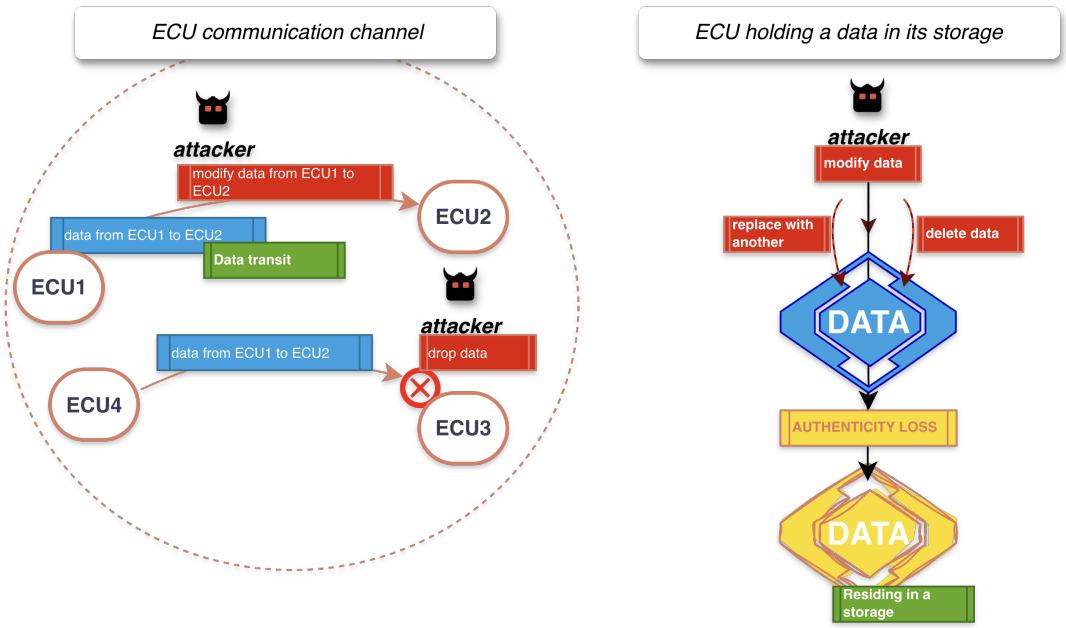

Figure 2.4: Integrity protection for data at rest or transit

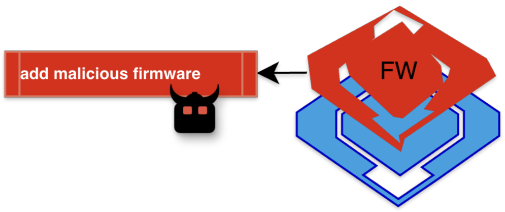

Figure 2.5: Attack modification of FW

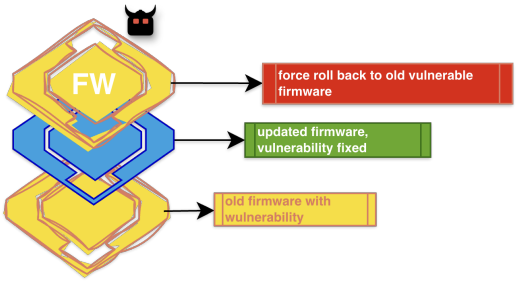

Figure 2.6: Attack rollback to old FW

of origin and must be authentic. Therefore, the care is not only on the data but also about who created that data. In order to check both of these properties, in standards the digital signatures are preferred. This is because if either the data itself or the originator is not legitimate then the verification will fail. And the digital signatures can provide authenticity and non-repudiation. Note that the protection of the firmware is taken as case study in this thesis as demonstrating HSM's functionality. Because it is a crucial first step to care. Without the correct firmware the ECU will be compromised and can behave maliciously to perform the

actions in the figure 2.5 and 2.6. Essentially, a compromised ECU becomes serious threat for the vehicle. It can intercept the internal network traffic, act as a Man in The Middle (MITM) node, or behave maliciously.

Chapter 3

Architectural Platform & Simulation Environment

Building on the requirements that maps to the problem statement introduced in the Chapter 1, specific architectural platform and platform compatible simulation environment is selected. The decision is based on the tools that would allow realizing open platform, configurability, transparency, extensibility, threat modeling capability, timing and side channel analysis, architectural experimentation. Another point is the selection of specific core provided in platform that satisfies the automotive performance to design and implement the HSM. If the HSM is modeled as an MMIO protected peripheral component it requires a separate component to be modeled in the platform description. Otherwise, if HSM is tightly coupled component with ISA extension, a core that allows ISA extensions is preferred. This recent paper demonstrates *KeyVisor* [9], which uses ISA extension strategy for the cryptographic key protection. The architectural rationale of *KeyVisor* is structurally consistent with the custom opcode strategy. In order to realize these two setups, the environment information is summarized in the Table 3.1.

Host OS	macOS with UTM virtualization
Guest OS	Ubuntu Server
Pulp Chip	RI5CY [10]
Simulator	Pulp Platform GVSoC
Toolchain	PULP RISC-V Toolchain

Table 3.1: Environment Setup Information

The architectural pattern in PULP accelerator, is shaped around offloading computationally intensive or security critical tasks to dedicated hardware as seen

in the Figure 3.1 PULP components layout.

The implementation is structured into 3 stages to progressively achieve the defined security goals.

1. Strategy 1: Functional validation of PMP with MMIO modeled peripheral.
2. Strategy 2: ISA triggered offload, to reduce attack surface

3.1 RISC-V

RISC-V architecture provides an open-source ISA, it is therefore eliminates the dependence of proprietary processor specifications. Consequently, the open architecture allows instruction extensions that would otherwise be constrained by vendor definitions. If HSM tightly coupled with a custom ISA this architecture provides the essential requirement. In addition to extensibility, openness also provides architectural transparency. Which allows long term research by possibility of analysis of system-level threat modeling. Finally, to study process, memory system and security mechanism interaction becomes possible.

3.2 PULP

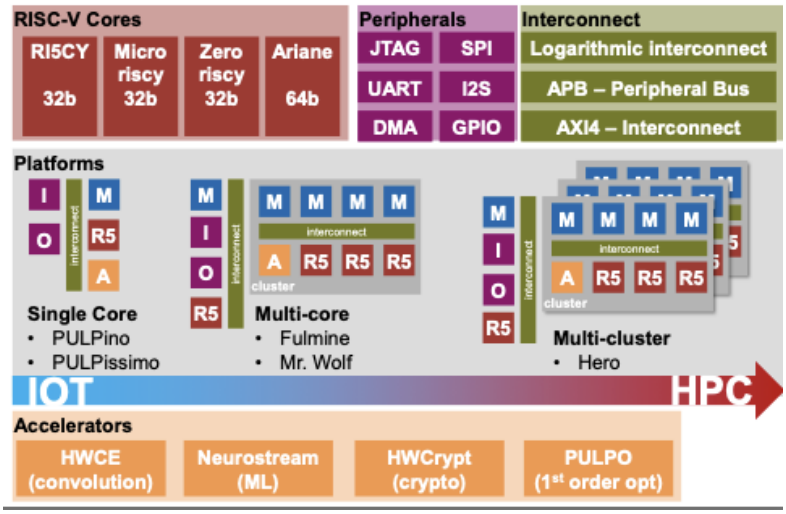

Figure 3.1: PULP components layout.

Parallel Ultra Low Power Platform (PULP) is providing flexible-system level architecture. Fully open and configurable SoC designed for RISC-V. Entire architectural stack is open including interconnect, memory hierarchy, peripheral integration.

Figure 3.1 sourced from [11] shows various core type and their use cases based on the performance considerations. It also illustrates the modular architecture of the platform with supported cores, peripherals and interconnects. It showcases the possibility to be able to scale the platform with multiple cores.

The development cycle utilizes the PULP-SDK as a reference and PULP-Toolchain. Pulp Chip used RI5CY [10], this is chosen specifically for the target automotive industry application.

3.3 GVSoC Simulation environment

GVSoC a cycle-accurate PULP chips simulator; in this work, it is emulating the chosen MCU core RI5CY, currently maintained by the OpenHW group [12].
In the research paper [13], GVSoC it is described as a 'Highly configurable, fast and accurate full-platform simulator for RISC-V based IoT Processors'. Based on this description, the mapping of each GVSoC feature this research utilizes is accordingly: *High configurability* is used for the RICV-V ISS modification, enabling protection flags, custom opcode implementation, RAM/ROM memory allocation, and interconnect wiring. *Accuracy* is important for threat modeling and attack simulations since it simulates the platform in a time-accurate way. It is cycle-accurate simulation environment for PULP chips. It is also attractive for the reasons that it allows for the creation of additional functionality with ISA extensions [13]. Performance of the GVSoC environment is studied in this paper [6] as mentioned in this work pulp-gnu-toolchain, is forked from the repositories.

RI5CY is supported by the PULP RISC-V GCC7 toolchain [14], targeting the rv32imc and extended variants. The target toolchain is affecting the ISA rv32imc.

3.4 RI5CY Core

There are variety of pulp chips available: Micro-riscy, Zero-riscy, RI5CY [10], RI5CY+FPU, Ariane. For the automotive embedded security requirements this work focuses and builds in top of RI5CY core.
From the documentation of the RI5CY core:

- 4-stage in order pipeline, this provides simple and analyzable execution model especially important for security experiments.

- Supports RV32I, RV32C, RV32M and PULP extensions which allows implementing and evaluating the custom HSM opcode inside the simulated environment.

- Includes performance counters through the CSR's. Therefore the counters for instructions, stalls, branch behavior can be observed. As a result the thereat modeling based on time observations is possible.
- Potential to activate a filter for the fetch, load and store requests executed in user mode using PMP flags.
- Dedicated instruction-fetch and load-store interfaces.
- Privilege support with machine and user status and exception CSRs. including mstatus, mepc, mcause.

This is because the automotive industry applications require a balance of performance capabilities and efficiency. Typically the security computations could create additional performance management necessities. Thus the HSM core should take the computational burden away from main CPU. RI5CY core has deterministic pipeline behavior and in order execution, use in embedded systems and energy efficient SoC. It supports custom ISA extensions. The slides *Understanding and working with PULP* [15] describes RI5CY as "workhorse core" of the platform which points the computational power as well as its extensive deployment across multiple silicon technologies.

3.5 Experiment Environment Setup Details

Strategy 1 focuses on the MMIO component orchestrating mycomp, mysystem, main.c and validating PMP support. Strategy 2, builds the HSM crypto features after the security validations of the strategy 1. This is because during the experiments at the strategy 1, it is observed that, the PMP protection is not activated with the ISS simulations.

Evaluation of possible design strategies under the experimental setup:

- Strategy 1: HSM function as a protected MMIO peripheral.
- Strategy 2: HSM function as a standard memory, mapped input output, peripheral HSM with tightly coupled ISA extension strategy

PULP-SDK provides some ready test for target platforms i.e., pulp-open. A simple target platform with single core is constructed. Similar to the provided GVSoC developer tutorials [1]

[1] `in GVSoC core docs/developer_manual/tutorials/1_how_to_write_a_component_from_scratch`

Used tutorial 11 from the GVSoC library and in the `my_system.py` several modifications were made to adapt the system to 32-bit architecture. Specifically the isa type was changed from isa='rv64imafdc' to isa='rv32imafdc' and the class was renamed from Rv64 to Rv32.

```
host = cpu.iss.riscv.Riscv(self, 'host', isa='rv32imc', binaries=[
    binary])
```

instead of Rv64 used Rv32

```
description = "HSM extension"
    model = Rv32
    name = "test"
```

In the Makefile used –vcd –trace=level=debug –power which generated the necessary gtkwave files that allows for future experiments analysis. This is in general important to measure for threat modeling and evidence for compromise:

- Cycle counts
- Execution latency
- Bus contention
- DMA throughput
- Timing side channel
- Custom instruction latency

Chapter 4

Methodology & Architectural Design

Different architectural designs can realize an HSM functionality, each comes with their own design constraints, trade-offs, and the potential practical limitations. The objective is to implement an isolated computational capability for the cryptography, key protection and controlled interface to invoke HSM. Therefore the design strategies were first; that is classic separate and protected component model and second the ISA extension with custom opcodes, that is, conversely to the isolation based on the address space, it provides isolation through its execution path. In the following two sections the architectural differences between the two strategies. Evaluation of each strategy with respective experiments presented after the design explanations.

4.1 Design Strategy 1: HSM as MMIO component

This design implements HSM module as an external memory mapped peripheral, interfaced through the main SoC interconnect and accessible through a dedicated protected MMIO region, access is specifically mediated by privilege and isolation mechanisms. Start of design with this peripheral model architecture for HSM, also allowed platform validation since the tutorials provided by GVSoC was changed specifically to satisfy the RI5CY. With strategy 1, thus aimed us to verify a correct component mapping, address recording exercise, software to hardware command interface, register protocol, status, interrupt signaling error paths and explore the platform available security relevant controls, privilege checks, access control,

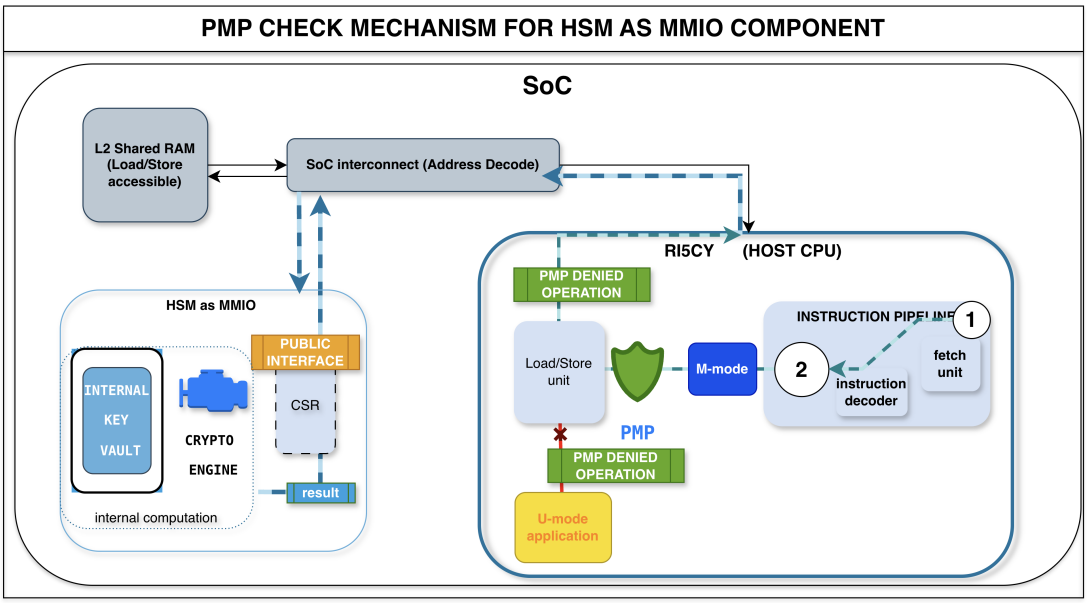

Figure 4.1: PMP protects against low privileged user mode access to HSM internal key vault

configurable flags, and integration points for protection mechanisms. One such protection is the RISC-V Physical Memory Protection (PMP) unit. It enforces access control over memory region based on permissions for (R)ead/(W)rite/(E)xecute across privilege levels. Figure 4.1 memory access flow through the PMP unit and HSM MMIO interface. The instructions fetched and decode are subject to PMP evaluation before the bus access. Access is granted to machine mode and denied for user mode. A violation of this these permissions creates an abnormal condition that creates a *trap handle* situation. When a CPU performs an emergency jump to a privileged handler in an abnormal situation, this is called **trap**. The abnormal situation can be a forbidden memory access or illegal instruction. If a trap happens CPU switches to M-mode, jumps to address held in *mtvec* and records both he cause and the location of the fault.

Even though this architecture model of HSM has interface separation it has limitations. The HSM block is not a fully isolated security domain by itself. This is not due to an absence of a dedicated core but because there is no prevention issuing MMIO read/write to an address range from compromised core no protection like hardware firewall/bus matrix rules. There should be an access control based on privilege for who wants to use HSM restricted to M-mode enforcing that only trusted firmware can issue commands to the module. Thus in the absence of isolation guarantees enforced by PMP any sufficient privileged or compromised

software executing in the main CPU, including U-Mode code retains the ability to directly access the HSM MMIO interface.

4.2 Design Strategy 2: HSM as Custom ISA Extension

HSM primitive for secure boot realized as a custom ISA extension is tightly coupled with RI5CY core. In contrast to the first strategy the SoC interconnect, L2 RAM and MMIO interface are entirely bypassed as observed in the Figure 4.2. Instead, all HMS related functionality is entirely internalized withing the custom security opcode. This tight coupling removes the bus fabric and reduces the attack surface. Since no HSM related transaction is externalized over the system interconnect.

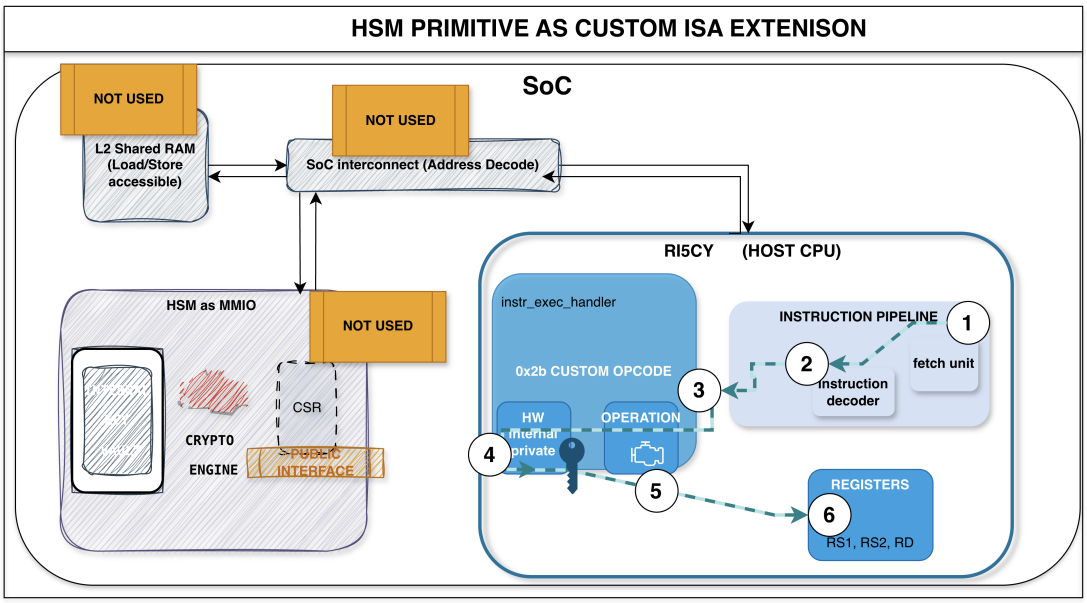

Figure 4.2: Custom security opcode is invoked in RI5CY

4.3 MMIO component PMP validation experiment

Among the two strategies mentioned the HSM as a MMIO peripheral was chosen to be the first to be implemented in the baseline system on chip. The reason why

is to avoid breaking the ISS by modifying the ISA without proper understanding of the component model, understanding of the platform configuration.

This prototype becomes starting point to experiment with the actual platform configuration, therefore allows observing in the PULP chip components.

4.3.1 Technical Words and Their Meaning

The configuration required setting up the values of Control and Status Registers (CSRs) that are referenced from RISCV manual [16].

- **CSRR** : Control and status register read
- **CSRW** : Control and status register write
- **mstatus** : Machine Status Register 0x300 (CPU privilege modes and global state flags)
- **mtvec** : Machine Trap-Vector Base-Address Register 0x305 (Trap handler lives here)
- **mepc** : Machine Exception Program Counter Register 0x341 (when the trap occurred where was the CPU)
- **mcause** : Machine Cause Register 0x342 (Shows the cause of trap)
- **mtval** :Machine Trap Value Register 0x343
- **pmpcfg0** : 0x3A0 Physical memory protection configuration
- **pmpaddr0** : 0x3B0 Physical memory protection address register with the number.

User mode (U-mode) is the lowest privilege level,

- access permitted to only PMP allowed memory regions.
- access to restricted subset of CSRs.
- trigger exception through *ecall* to request M-mode services.
- execute unprivileged instructions only.

Machine mode (M-mode) is the highest privilege level for a SW with capabilities of and not limited to :

- unrestricted access to the memory, operates without PMP restrictions.
- configure and access to all CSRs.
- handle exceptions and interrupts.
- control boot flow.

The following values of the CSR's and the values they take depending on the scenario is referenced from the RISC-V instruction set manual Section3.7 with more [17] extensive details. One important reduced table for mcause CSR value with its description is presented with the table from the manual in Figure 4.4.

Each entry in the RISC-V PMP, has an address that is pmpaddrN and a config byte pmpcfg. The permissions are set in the config with R/W/E.

Algorithm 1 PMP enforcement verification experiment in GVSoC. Experiment denies access to a selected memory region and checks if user-mode load operations trigger a trap. The outcome of the CSR mcause register and MMIO based debug markers show if load is blocked or not

procedure PMP ENFORCEMENT CHECK(
    ▷)MYCOMP_BASE is a memory-mapped debug peripheral
    ▷ TEST_BASE is a memory-mapped debug peripheral
    MYCOMP_BASE ← `0x20000000`
    TEST_BASE ← `0x00020000`
    TEST_SIZE ← `0x1000`
    ▷ Trap related CSR's
    mstatus ← `0x300`
    mtvec ← `0x305`
    mepc ← `0x341`
    mcause ← `0x342`
    mtval ← `0x343`
    ▷ PMP CSRs
    pmpcfg0l ← `0x3A0`
    pmpaddr0l ← `0x3B0`
    ▷ Initialization
    CSRW(mtvec, &TRAPHANDLER) ▷ Install machine mode trap handler
end procedure

procedure UNTRUSTEDUMODE
    MMIO[MYCOMP_BASE + 0] ← `0x55555555` ▷ Debug flag: Entered to U-mode
    x ← LOAD32(TEST_BASE) ▷ Should trigger PMP fault if enforcement is active
    $xMMIO$[MYCOMP_BASE + 0] ← `0x66666666` ▷ Printed only if PMP is NOT enforced
    $xCSRR$(mstatus) ▷ Illegal CSR asccess in U-mode, expected to trap
    Loop forever
end procedure

The pseudocode with the **Algorithm** 1 shows the experiment logic of the MMIO component with certain flags that allow us to interpret the execution flow.

This experiment validates whether PMP is correctly enforced by attempting an unauthorized memory access from an unprivileged CPU mode and observing whether the expected fault occurs.

- *0x11111111* and *0x22222222* indicates program initialized , **mtvec** trap handler correctly configured.
- *0x55555555* is the first flag to indicate that the CPU dropped to the user mode. In this low privilege mode and untrusted state, load from denied region is attempted.
- *0x00000002* mcause is equal to value 2, confirming an illegal instruction exception. It is used as an additional check to verify the CPU is genuinely in U-mode. Since *0x55555555* alone does not guarantee this.
- *0x0000000* is reported from the trap handler,following an illegal CSR read attempt in U-mode where mtval = 0 indicates the bad instruction value.
- *0x00000005* mcause is equal to value 5, confirming PMP is enforced correctly.

The consideration about option 1 before starting the experiments was clear. If

```
static inline uint32_t read_mstatus(void){uint32_t x; __asm__ volatile("csrr %0, 0x300":"=r"(x)); return x;}
static inline void write_mstatus(uint32_t x){__asm__ volatile("csrw 0x300, %0"::"r"(x));}
static inline void write_mtvec(uint32_t x){__asm__ volatile("csrw 0x305, %0"::"r"(x));}
static inline void write_mepc(uint32_t x){__asm__ volatile("csrw 0x341, %0"::"r"(x));}
static inline void write_pmpaddr0(uint32_t x){__asm__ volatile("csrw 0x3B0, %0"::"r"(x));}
static inline void write_pmpcfg0(uint32_t x){__asm__ volatile("csrw 0x3A0, %0"::"r"(x));}
```

Figure 4.3: Setting up CSR values

the HSM is built without extra isolation mechanism, it will be insecure due to the vulnerability to key leakage, Direct Memory Access (DMA) snooping, bypass of privilege mechanism. In the RI5CY core documentation one memory protection mechanism and the way to enable it was presented. During the experiments the PULP_SECURE macro was set to the value 1 as required.

Table 16. Machine cause (`mcause`) register values after trap.

Interrupt	Exception Code	Description
0	2	Illegal instruction
0	5	Load access fault

Figure 4.4: mcause CSR's exception codes

4.4 Functional Validation with MMIO Component

At the start SoC level integration is validated by setting up a prototype MMIO component. This is similar to the orange colored component in the platform Pulpissimo (pulp-open) as presented in the Figure 3.1, instead we have custom platform (my_system.py). Interconnect routing. Checking if the memory and MMIO peripherals are mapped correctly. Instantiate the RISC-V ISS (RiscvCommon) with enabling the user mode and PMP.

Platform wiring is set in the mysystem.py is using gapy to configure the python based custom target. Another possible configuration can be done by JSON config files as introduces in json config [18] github page.

For this validation as well as for the stage 2, library provided utility files are accumulated in the `gvsoc_clean/thesis_utils/`

In the pseudo code of the experiment 1 it is expected that, REG0 showing a load access fault with mcause=5, REG4 to have 0x0020000 and CPU to be stuck in the handler loop, if after the run 0x66666666 is obtained on REG0, this means PMP did not block.

```
1    host = RiscvCommon(self,      host     , mmu=True, pmp=True,
    fetch_enable=False, boot_addr=0, supervisor=True, user=True)
```

4.5 Experiment Setup: Secure boot with ISA Custom Opcodes Extension

ISA extension is deep in the GVSoC library thus extension requires modifications by adding custom opcodes and understanding how an instruction is fetched/decoded/executed. To get used to library experiment starts with the goal of implementing a *simple instruction* that performs a math operation to validate the ISA extension

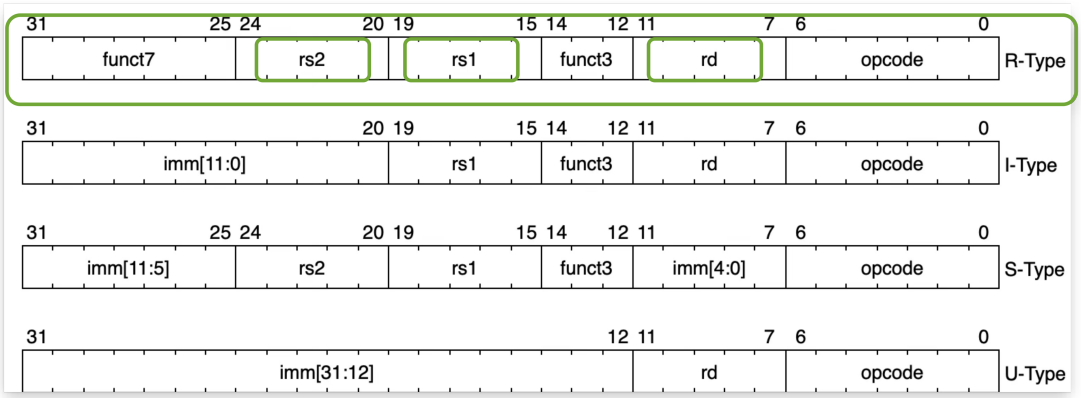

Figure 4.5: Selected custom instruction type and its structure

with additional ISS. Created a copy of ISS extension form GVSoC developer manual tutorials. to modify and integrate with the system. Inside the GVSoC library the `models/cpu/iss/isa_gen/isa_riscv_gen.py` has other instructions for rv32i ISA. ISA was expanded with custom instruction *my_instr*. The specific extension format is R-type instruction. As depicted in 4.5 obtained from the source [19]. The code snippet show exactly how the custom instruction inside Rv32i(IsaSubset) located in the isa_riscv_gen.py of GVSoC library was defined.

```
class Rv32i(IsaSubset);
    def __init__(self):
        super().__init__(name='rv32i', instrs=[ ...
Instr('my_instr', Format_R, '0000000 ----- ----- 000 ----- 0101011')
    ]
```

The 4.6 depicts the decoding of custom ***my_instr***. The last 0101011 is the opcode which identifies this instruction as custom-0. The instruction was invoked within the software toolchain by the 32-bit word displayed in the Figure 4.6 asm_instr.S.

From the figure 4.5 we can map the specific fields values, i.e. **0000000** is func7, **01011** is rs2, **01010** is the rs1. Significant field values for later stages are **rs1, rs2, rd**. They are the first, second argument and the return value respectively.

Below is the exec-handler, which performs the addition and multiplication, to rv32i.hpp [1]

[1]At the time of writing this work located in `models/cpu/iss/include/isa/` in GVSoC core

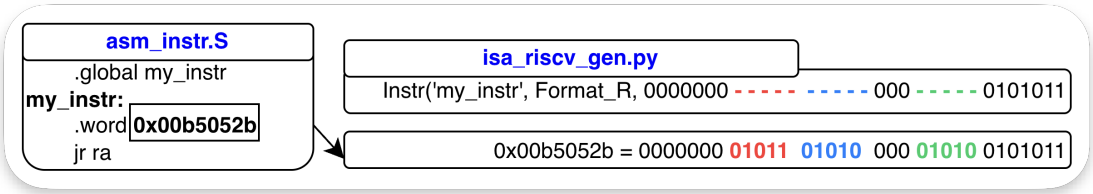

Figure 4.6: Custom instruction and it decode

```
1    static inline iss_reg_t my_inst_exec(...)
2    REG_SET(0, REG_GET(0) + 2 * REG_GET(1));
```

after the build and the run build, install and test artifacts are generated in the

```
/home/ozge_pulpvm/pulp_workspace/gvsoc_clean/_work/tutorial_hsm/install/bin/gvrun \
        --platform=gvsoc \
        --target-dir=/home/ozge_pulpvm/pulp_workspace/gvsoc_clean/core/models/devices/tutorial_hsm \
        --target=my_system \
        --work-dir=/home/ozge_pulpvm/pulp_workspace/gvsoc_clean/_work/tutorial_hsm/work \
        --param=binary=/home/ozge_pulpvm/pulp_workspace/gvsoc_clean/_work/tutorial_hsm/test/test \
        --py-stack \
        --verbose=debug \
        run
Launching GVSOC with command:
gvsoc_launcher --config=gvsoc_config.json
10 + 2 * 5 -> 20
```

Figure 4.7: ISA extension functional verification with simple arithmetic operation

`home/ozge_pulpvm/pulp_workspace/gvsoc_clean/_work _work/tutorial_hsm/iss/include/ isa/`

After the verification of simple custom instruction is working and integrated to the GVSoC ISS, the next step was to use the same strategy but transform this instruction to be part of secure boot prototype. Similar to the earlier stages of the experiments, instead of implementing the most secure algorithms and standards the goal instead is to ensure the security flow with respect to the program. Finally to have a solid base to replace with a desired crypto algorithm and mode. To this end, the primitive secure boot is strted and implemented.

Turning Simple Custom ISA to Secure Boot Verification

Figure 4.8, provides a high-level overview of the complete automotive boot sequence, covering initial power on to the application handover. At power on, phase 0 the ROM bootloader initiates the primary boot loader establishing the first executable context on the ECU. Phase 1 encompasses the first stage booth verification, after which phase 2, introduce the decision point depending on the

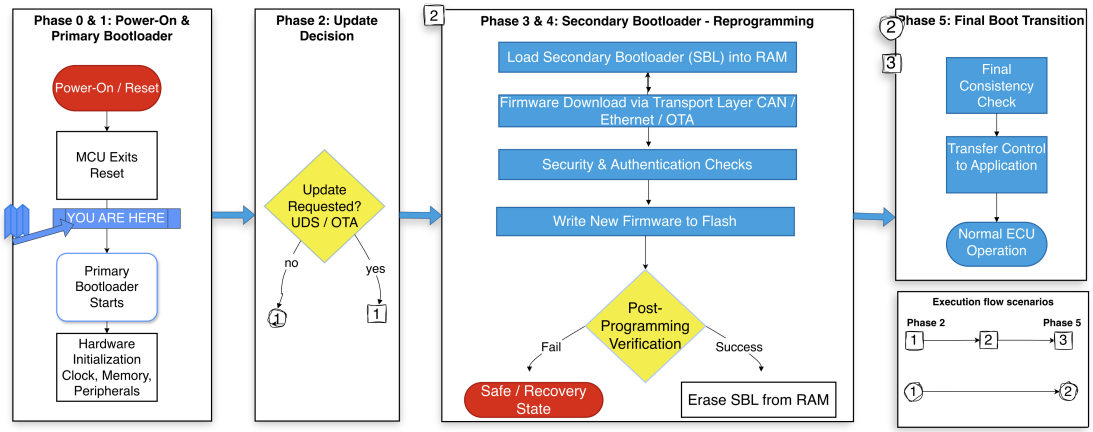

Figure 4.8: ECU boot phases high level

presence of any update. If the scenario update request present, then the ECU cannot simply go to phase 5, it must go through several steps namely the reprogramming performed by the SBL. Phase 3 of the Figure 4.8 is located in the mutable memory area and following steps are performed in order. Load the SBL, download firmware via transport layer, this can be by OTA update, Ethernet or by UDS diagnostic services over CAN bus. Following this step, security and authentication checks are performed.
Before reaching the post programming verification step new firmware is written to the flash. If the post programming verification succeeds the second bootloader is erased from RAM, and ECU can finally transition to phase 5, where final boot transition takes place and after its completion ECU starts its normal operation.

The security phases demonstrated in the Figure 4.8 of secure boot is pictured using the information inspired from the [20].

The 4.8 presents a high level view of the ordered process taken before reaching normal operational ECU. However, to understand how a hardware root of trust can be emulated within a simulation environment such as GVSoC, the focus must be narrowed down to a precise segment withing the Phase 0 and 1. The 4.9 is the detailed process description of the implementation of second architectural strategy. Specifically where custom ISA extension for signature verification is present in the full picture.

Specifically, the point where it shows "you are here" in the full secure boot.

The implementation emulates the HRoT behavior and early stages of secure boot chain. This process is illustrated in the Figure 4.9. If we take the process map location and map this stage to the Figure 4.8 it corresponds to the stage in between the MCU exit reset primary bootloader starts.

Assumption: The 4.9 is showing green dashes for GVSoC simulator execution

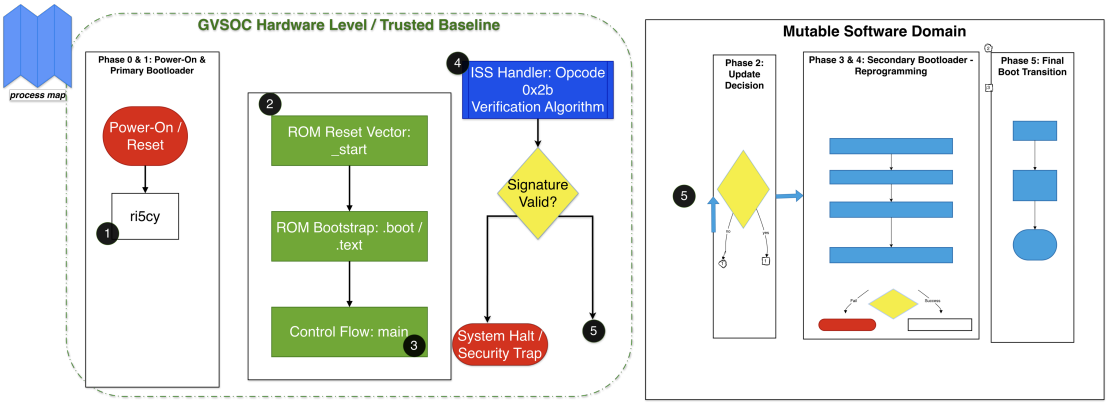

Figure 4.9: RoT functional implementation logic in GVSoC

logic, which implies our trust in its correct functioning as a baseline. In real silicon experiments trust to GVSoC as environment is a similar assumption as functional and micro-architectural correctness of the physical hardware platform.

Execution: The code execution starts with **_start** and **main**, both are resident in ROM. This is the security policy. The transition flow from step 3 to 4 illustrated with the Figure 4.10. It is essentially passing from ROM's trigger to execution of our custom opcode 0x2b. We therefore achieve the instructions from ROM with an execution logic of GVSoC. ISS handle implemented in the rv32i.hpp is implements the verification algorithm as hardware primitive. The transition to the mutable code happens on step 5, when the ISS opcode 0x2b returns *success* to **main**, In this setup the Boot ROM is emulated with the ROM resident boot code. Boot payload is then stage 1 bootloader dummy boot image in the main. It corresponds to firmware image received from non volatile storage in physical word.

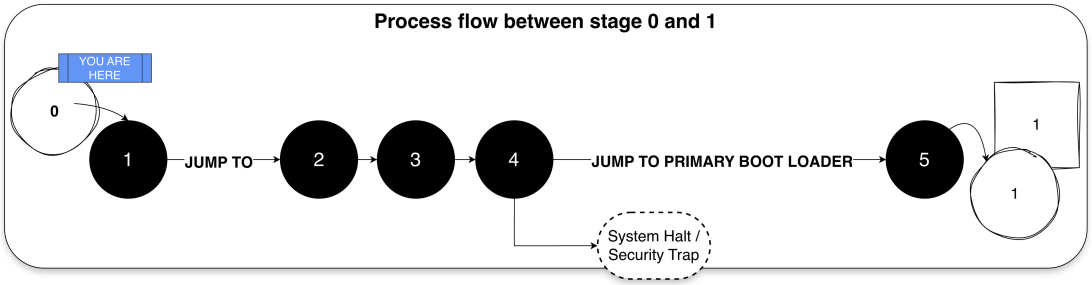

Figure 4.10: Process flow steps secure boot phase 0

Figure 4.10 illustrates the flow order in performing low-level implementation details between the phase 0 and phase 1. Where boot image is verified with the ISA opcode triggered from ROM.

The earlier GVSoC experiment involving the simple arithmetic my_instr in Section 4.5 instruction did not enforced the use of immutable memory region for code execution. After verification of ISA extension success, the transformation of the initial arithmetic opcode prototype into a secure boot verification mechanism requires the introduction of additional architectural structure inside the GVSoC platform. Even if the simulation environment does not provide a physical HSM with One Time Programmable (OTP)/eFuse for key storage as explained in the physical HSM component 2.1, the emulation of key protection in this practical constraint environment is as follows.

First **secure boot phase 0** begins by defining a dedicated immutable memory region that represents the system's boot ROM. This region, serves as the emulated hardware root of trust and contains the boot code responsible for performing firmware verification. The build configuration and linker are explicitly arranged so that the program and entry point is located within the read only memory region. The processor begins execution from the trusted ROM address space after reset, allowing that verification routine in every ECU boot is executed before any code located in mutable memory is allowed to run.

The verification stage of the secure boot chain implemented using simplified symmetric cryptographic operation that allows the interaction between the processor and the HSM instruction handler to be functionally exercised. Rather than implementing a full public key signature scheme. The current model uses a deterministic mixing function symmetrically to represent the verification step. The purpose of this choice is not to claim cryptographic strength, but to focus the evaluation on architectural behavior:

- Invocation of the HSM operation.
- Protection of internal state.
- Time characteristics observable at the processor interface.

Even though production automotive systems rely on asymmetric signature schemes such as ECDSA, incorporating a complete implementation at this stage would not provide additional advantage into timing site channel and fault injection modeling implemented in this work. Instead the simplified operation allows the verification flow and attack models to be studied without introducing a substantial, computational complexity of full public key cryptography. If a production grade implementation be required the internal verification logic of the instruction handler must be replaced with a standard public algorithm with reusing of boot ROM configuration to store public key.

Among the provided tutorials provided in the GVSoC development manual[2],

[2]located in `pulp_workspace/gvsoc_clean/core/docs/developer_manuals/ tutorials/`

one is about how to add a new instruction. It is used as a starting point to turn the current simple custom operation to the PBL verification of SBL thus necessitation a ROM region in addition to the existing RAM. As immutable region is the location where the execution starts. The Figure 4.11 displays the address range we allocated for the immutable first sage storage ROM and the mutable storage RAM.

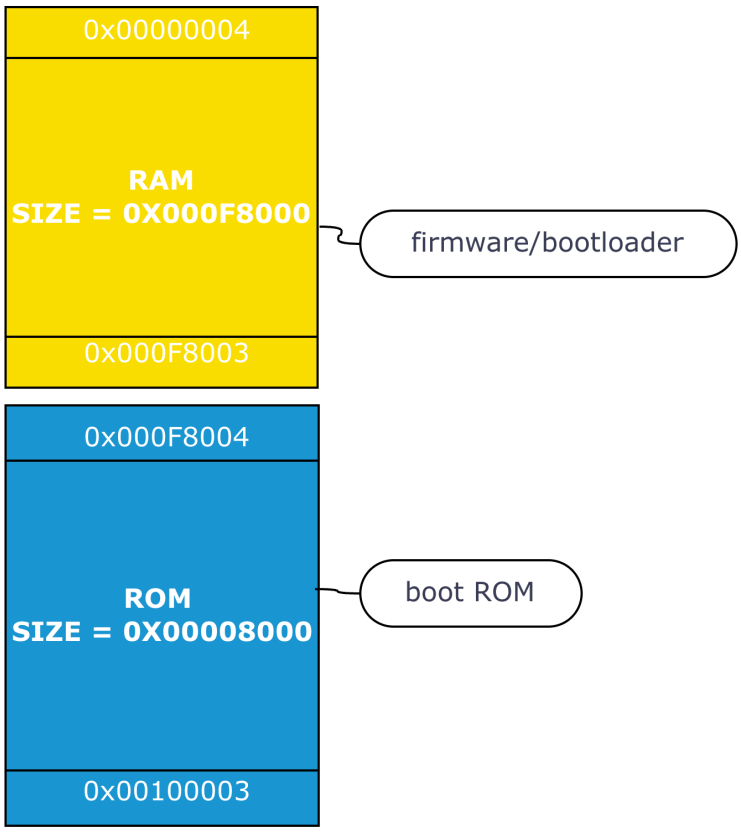

Figure 4.11: Memory partitioning to RAM and ROM regions

Listing 4.1: RAM/ROM memory partitioning

```
1    mem_ram =memory.memory.Memory(self, 'ram', size=0x000F8000)
2    ico.o_MAP(mem_ram.i_INPUT(), 'ram', base=00000000, size=0x000F8000
     , rm_base=True)
3    mem_rom =memory.memory.Memory(self, 'rom', size=0x00008000)
4    ico.o_MAP(mem_ram.i_INPUT(), 'ram', base=000F8000, size=0x00008000
     , rm_base=True)
```

`11_how_to_add_an_iss_instruction` in GVSoC core

Note that or alignment and avoiding placement of data/code at address 0x0 there is a 4 byte offset. The actual RAM region spans the address 00000000 to 0x000F7FFF, and the ROM region is 0x000F8000-0x000FFFFF. Below implementation is the previous memory that is the writable RAM, it mirrors the linker script as shown in the 4.12.

Before there was on only the RAM as the main memory.

Listing 4.2: RAM memory and no ROM partitioning

```
1  mem =memory.memory.Memory(self, 'ram', size=0x00100000)
2  ico.o_MAP(mem_ram.i_INPUT(), 'ram', base=00000000, size=0x00100000
   , rm_base=True)
```

In the code snippet 4.5 the implementation of the ROM partitioning is demonstrated. As a result, the memory is split into two distinct regions: a mutable RAM and an immutable ROM. Moreover the R/W/X permissions set are observable in the Figure 4.12.

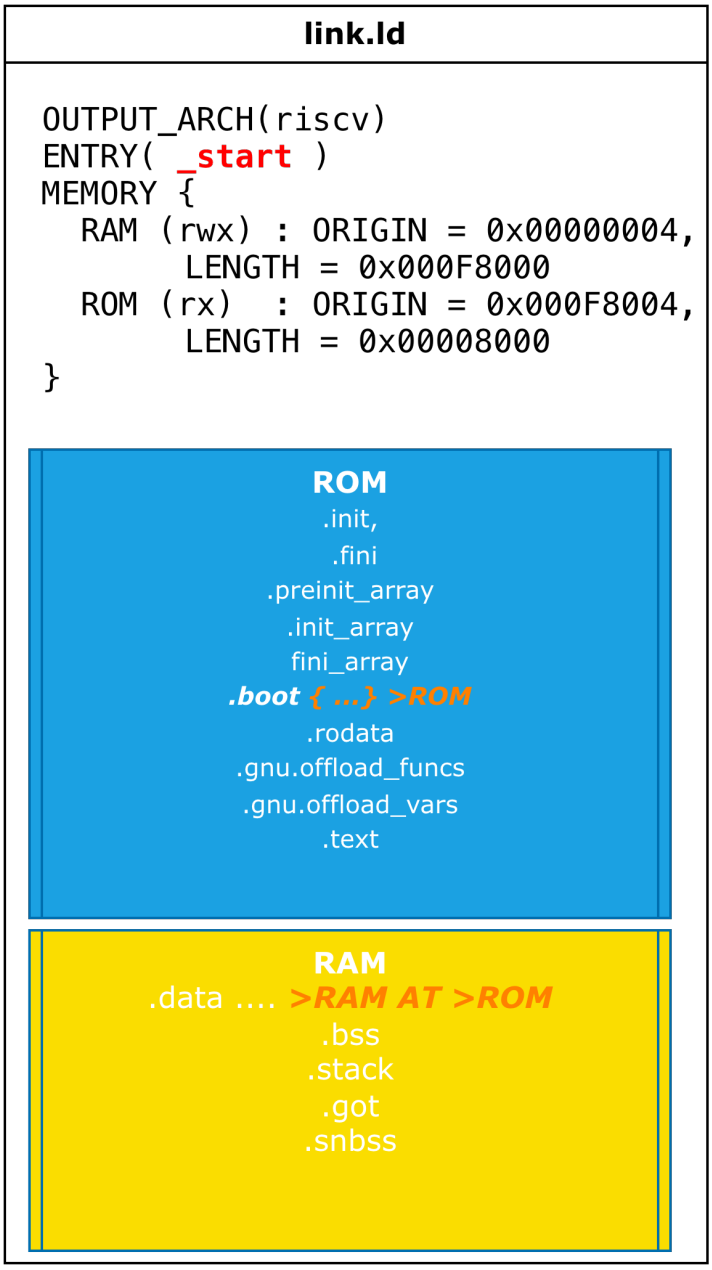

Figure 4.12: Memory sections partitioning

4.5.1 Ensuring execution from BootROM

```
1    loader = utils.loader.loader.Elfloader(self, 'loader', binary=
     binary, entry=0x000F8028)
```

Force start at ROM region is achieved with the code snippet snippet 4.5.1 the `entry=0x000F8028` is not arbitrary, it is the address of the _start that is proved to be in the partitioned ROM region as observed in the output of the address verification shown in the Figure 4.14.

crt0.S located in `gvsoc_clean/thesis_utils/` is the runtime startup file that defines the reset entry point . [3] It defines the _start as show in the process 4.17. Then the linker script must have the .boot defined in ROM. We demonstrated the implementation of memory sections in the 4.12 and the in the following sections the verifications of their placement in memory regions is provided.

The _start is the first instruction to be executed as reset/entry as the linker uses ENTRY(_start) it corresponds to the Boot ROM entry and jumps to the ___init_start from the init.c, that invokes the main() from ROM that is simulating [4] first stage secure boot verification triggering the custom ISA opcode 0x2b.

For the purpose of the simulation, GVSoC is a trusted baseline and it's underlying architectural emulation is assumed to be an accurate and secure representation of the intended hardware behavior.

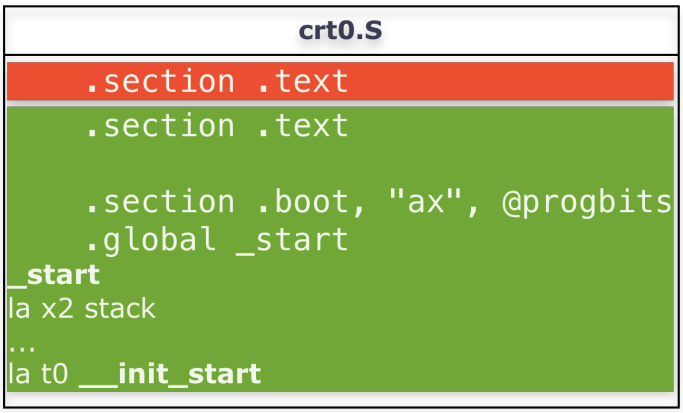

Figure 4.13: Changes made to crt0.S file

The 4.13 shows the changes made to the original crt0.s file for link time placement. **__start** is forced into **.boot**, and .boot is forced into . Figure 4.12 displays how the memory sections are assigned to RAM or ROM. Security policy including the

[3] the original of the file can be seen in the `docs/developer_manual/tutorials/utils/crt0.S`

[4] ROM is implemented as a memory object in GVSoC not hardwired logic in a silicon.

boot-strap sequence and control flow is hosted in the carved ROM region. To verify this we checked the following conditions :

- **__start** is placed in ROM
- **crt0.S** is placed in ROM
- **main** is linked into ROM

Figure 4.14: Evidence for **main**'s presence in ROM

Verification of where exactly the **main** is located is done as seen in the Figure 4.14. Since the main in the address 0x000FB21A and the rom is between 0x000F8004 -> 0X000FFFFF main is therefore linked to the ROM.

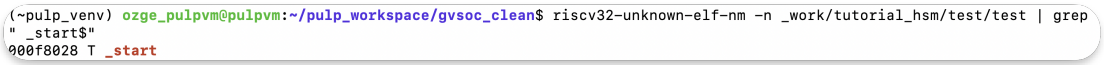

Figure 4.15: Evidence for **__start**'s presence in ROM

Similarly, Figure 4.16 is displaying the verification of the addresses for .boot .rodata. and .text address locations where, reset entry **.boot** and early bootstrap is in ROM Constants and immutable data in ROM, by setting **.rodata** to ROM. As well as the *main()* executable code corresponding to **.text** is located in ROM.

This verifications steps helps us observe that system is booting directly into an immutable ROM region. The visual flow of build process is depicted with the Figure 4.17.

```
(~pulp_venv) ozge_pulpvm@pulpvm:~/pulp_workspace/gvsoc_clean$ riscv32-unknown-elf-readelf -S _work/tutorial_hsm/test/test |
egrep '\.boot|\.text|\.rodata'
  [ 6] .boot             PROGBITS        000f8028 001028 000038 00  AX  0   0  1
  [ 7] .rodata           PROGBITS        000f8060 001060 000510 00   A  0   0  4
  [15] .text             PROGBITS        000f8570 002570 002d90 00  AX  0   0  2
```

Figure 4.16: Evidence for memory sections **.boot**, **.rodata**, **.text** presence in ROM

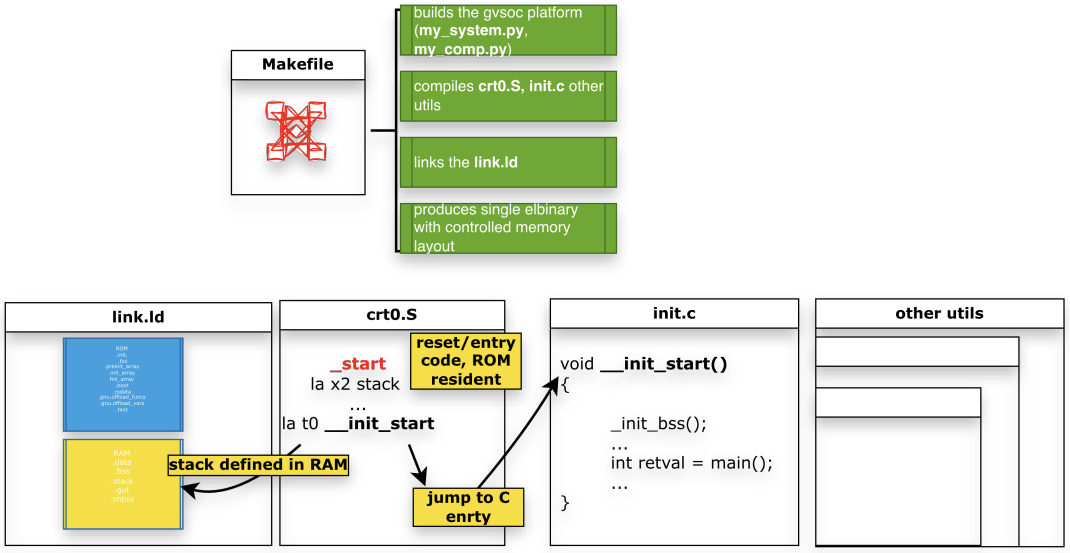

Figure 4.17: Build process

4.5.2 Custom Opcode Signature Verification Implementation Details

In the subsection 4.5 the custom instructions execution handler performed a simple math operation. After the functionality as shown in the Figure 4.7 was verified, the transformation of the execution handler to a secure boot verification handler is performed. The complete code section is present in the appendix. It is useful to map the simple execution details mentioned in the previous subsection 4.5 important fields of the custom instruction. After transformation, *rs1* holds the **boot digest**, *rs2* is the provided **signature** and the signature verification result is set to 1 if verified successfully, 0 otherwise.
It is important to explicitly acknowledge the cryptographic simplification made in the implementation. Technically it is behaving like keyed digest rather than asymmetric cryptography, where in practice, signing is done with OEM's private key and the verification is done OEM's public key. *Crucially note that this implementation is neither acceptable for the non-repudiation since the symmetric key*

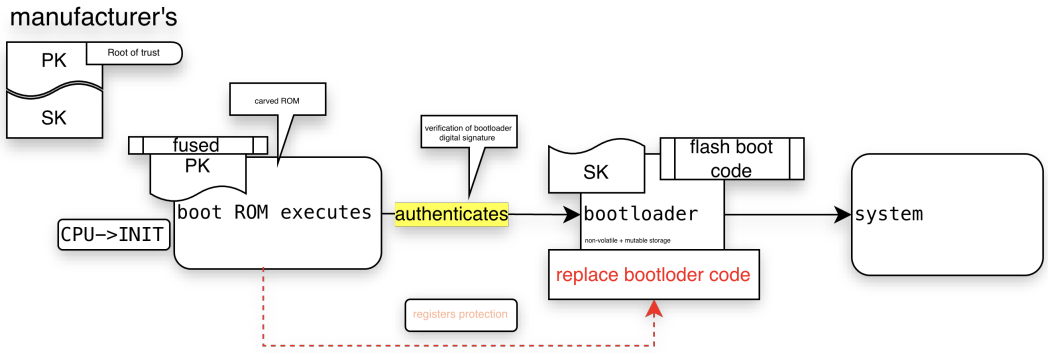

Figure 4.18: Bootloader verification, conceptually adapted from the book[8] pp. 5. SK=Private Key PK=Public Key

*cryptography **cannot** provide legally undeniable proof of origin this property nor it is in parallel with current cryptography standards.* This abstraction is done only for the functionality verification of secure boot flow and efficiently represent the side channel leak in the threat modeling.

Inserting source code added to [5]

```
1   static inline uint32_t simple_verify(uint32_t digest
    ){
2       const uint32_t fused_key = 0xa5c3f18du;
3       uint32_t mac = digest ^ fused_key;
4       mac = (mac << 7) | (mac >> 32 -7);
5       return mac ^ 0x3c6ef372u;
6   }
7   static inline uint32_t my_instr_exec(Iss *iss,
    iss_insn_t *insn, iss_reg_t pc){
8       const uint32_t digest = (const uint32_t)REG_GET
    (0);
9       const uint32_t provided = (const uint32_t)
    REG_GET(1);
10      const uint32_t expected = simple_verify(digest);
11      const uint32_t verified = expected == provided;
```

[5] the path `pulp_workspace/gvsoc_clean/core/models/cpu/ iss/include/isa/` in GVSoC core

```
12      REG_SET(0, verified);
13
14      }
```

main.c [6] controlled tampered signature

```
1      static uint32_t simple_digest(const uint8_t  *image,
       uint32_t len){
2           uint33_t acc =0x6d2b79f5u;
3           for (uint32_t = 0; i< len; i++){
4              acc^= ((uint32_t) image[i] << ((i & 3u) * 8u;
5              acc += 9x9e37799b9u;
6           }
7           return acc;
8   }
9
10          static inline uint32_t sign(uint32_t digest){
11          const uint32_t fused_key = 0xa5c3f18du;
12          uint32_t mac = digest ^ fused_key;
13          mac = (mac << 7) | (mac >> 32 -7);
14          return mac ^ 0x3c6ef372u;
15      } //the simple mixing just mimic signing
16
17      int main(void){
18      static const unint8_t boot_image[] = {0x2c ,.... }
19      uint32_t digest = simple_digest(boot(image, sizeof(
       boot_image));
20      uint32_t signature = sign(digest);
21      uint32_t verified = my_instr(digest, signature)
22      printf("Secure boot opcode (0x2b) => %s\n", verified
       ? "PASS" : "FAIL");
23     uint32_t tampered= my_instr(digest, signature ^0x3u);
       //controled tampering
24      printf("Secure boot opcode (0x2b) => %s\n", tampered
       ? "PASS" : "FAIL");
25      return 0;
26      }
```

[6] the path `pulp_workspace/gvsoc_clean/core/models/devices/ tutorial_hsm/` in GVSoC core

4.5.3 Modeling Timing Based on Whitepaper

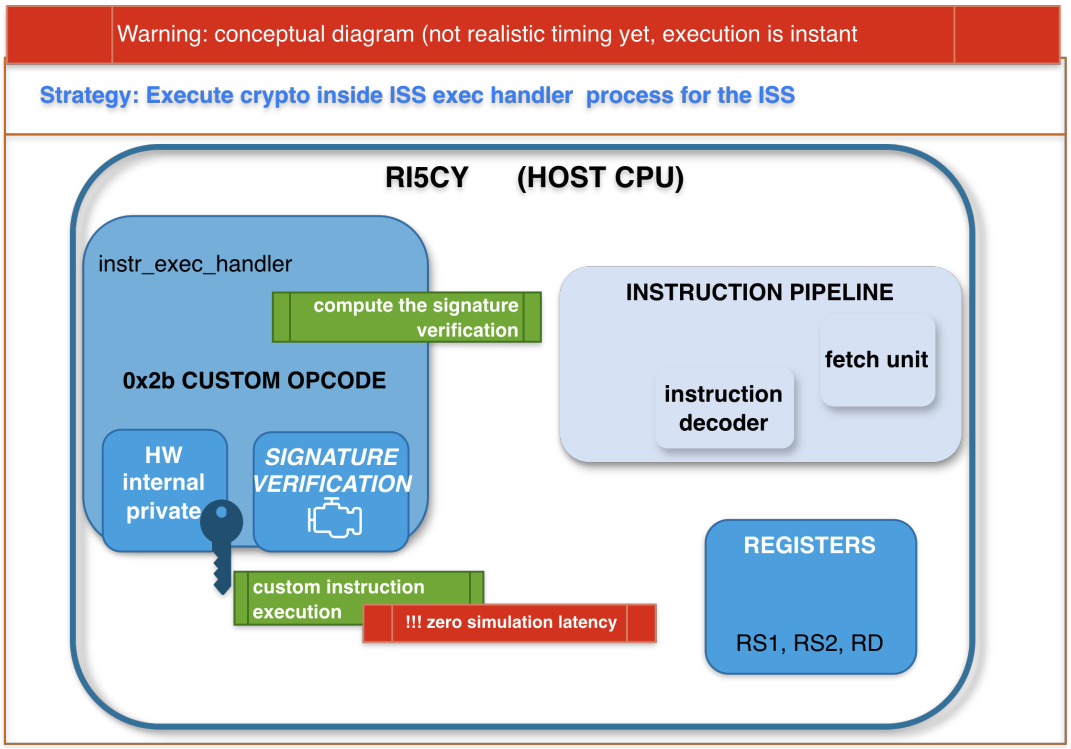

Figure 4.19: Relative vs absolute timing in security instruction execution

Although custom opcode performing secure boot signature verification primitive use simple mixing function with symmetric cryptography rather than the asymmetric cryptography, the stall cycle count in the *verification* is nonetheless aligned with the timing model studied in this whitepaper [21] Table 6, Section 7.1. This choice was also preferred based on the similarity of ISA type we use rv32imc as the ISA. They published cycle counts for similar cryptographic operations on RISC-V and embedded cores. That is, the embedded ECC signature verification routines have been measured to have several thousand cycles on small RISC-V processors in the cycle accurate simulation. Some definitions in the selected paper are; *rv32i* whihc is the software implementations using only base integer ISA, no dedicated crypto instructions or hardware functional units. rv32i + crypto instead uses crypto extension instructions (i.e. AES/SHA) using dedicated hardware units. RISC-V has also cryptographic extensions and they are considered rv32i+crypto instead of the base isa rv32i.

Below is a code snippet describing how to introduce custom stall cycles with

the example being 2 cycles.

```
1     iss->timinh.stall_cycles_account(STALL=2);
```

The artificial delay can be simulated by changing the STALL to a different value. The realistic time the signature verification takes is set to 2078 cycles, aligned with the SHA-256 *rv32i+crypto*. Execution latency reported by the study [21] report this exact figure as the clock cycle count for hardware accelerated SHA-256 on rv32i+crypto configuration on a 5-stage pipelined RISC-V processor.

rv32i.hpp time leak and realistic time modeled

```
1
2     const uint32_t digest = (const uint32_t)REG_GET(0);
3     const uint32_t provided_signature = (const uint32_t)
    REG_GET(1);
4     const uint32_t expected_signature = my_instr_mac(
    digest);
5     const uint32_t verified = expected_signature ==
    provided_signature;
6     REG_SET(0, verified);
7     iss->timing.stall_cycles_account(2708);
8     if(!verified){
9         iss->timing.stall_cycles_account(2708);
10    }
11    return iss_insn_next(iss, insn, pc);
```

Chapter 5

Threat modeling

5.1 Attacker Models

Figure 5.1 explains the attacker models with their mapping to a physical attacker, that is in the silicon SoC. The accompanying details justify which attackers in the threat model are considered to be applicable in this thesis. I.e., Sections 5.1.2 and 5.1.4. Otherwise, why it is not applicable as explained in detail in the Section 5.1.3, given the constraints of the architectural modeling and simulation environment circumstances.

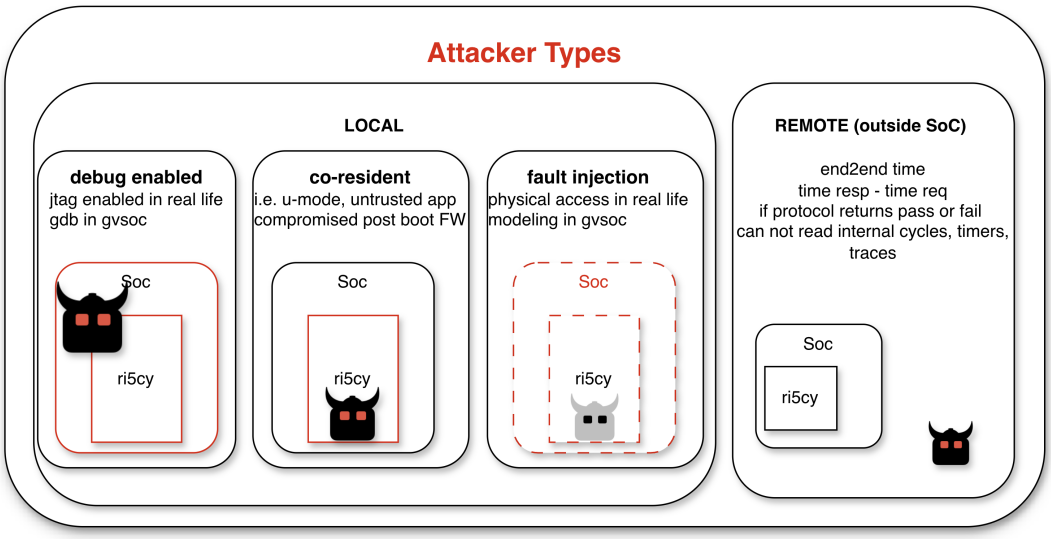

Figure 5.1: Attacker models

In the Figure 5.1, red continuous line considered risky areas, that are within the attacker's scope and capabilities. Red dashed lines are used for models that requires modeling to be demonstrated in simulation environment. Despite the deviations between the real and simulations environment fault injection symptoms functionally demonstrated. Below is a table that summarizes the status of attacker models in our threat model.

Local - fault injection	in the threat model
Local - debug enabled	out of scope for this work
Local - co-resident	in the threat model
Remote - outside of Soc threat model	in the threat model
Remote - power EM analysis	out of scope for this work

Table 5.1: Examples of attack models and its presence in threat model

5.1.1 Local Attacker Model Debug Enabled

As illustrated in Figure 5.1 under the label "*debug enabled*". An unlocked debug port exposes the system to an attacker capable of halting or manipulating execution flow directly through the debug interface.

Moreover it can measure timing/power/EM for side channels including the time side channel vulnerability. This attacker model is not considered in this study. This is because it is a total compromise. To support given justification, some examples of the following real systems are: NXP implementation [22], where the JTAG port is permanently disabled or accessed through authentication. Another similar example is from the ARM [23]. If the same reasoning is reflected to this study; considering debug enabled attacker, there would have been a serious inherent vulnerability due to the insecure debug interphase. Therefore, debug enabled attacker model is concluded out of scope.

5.1.2 Local Attacker Co-resident Model

This attack can be realized by means of untrusted code execution on the same core. An example could be the post-boot application software. One could argue that this is the software that is verified by the secure boot, secure boot protects against an attacker flashing a malicious boot image, but it does not mitigate post-boot software compromises. Another strong reason for us to consider this attack model is that PMP protection was not enforced. Consequently lack of privilege separation for executing load/store from untrusted U-mode. A possible scenario in the scope of timing leak side channel, is that untrusted code executing in U-mode, repeatedly

invokes our custom opcode because there is no difference between M-Mode and U-mode.

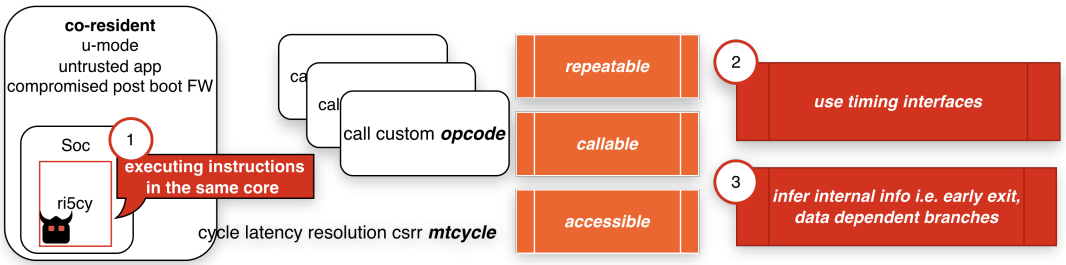

Figure 5.2: Local co resident attacker attack process

5.1.3 Power and EM Analysis

In real silicon, the attacker can measure the power trace, perform Differential Power Analysis (DPA)/Correlation Power Analysis (CPA)/Electromagnetic Analysis (EMA). With the objective of constructing secret key by observing this traces. Once the key is discovered then the attacker is authorized to perform the same actions of a legitimate signer. I.e, secure boot image signed however this time it will be malicious firmware. Since a silicon chip is emulated, the same observable traces is not expected. In fact this source [24] shows a capability of GVSoC for the power modeling. Notably, this is a model of power related to the activity and the cost it is not a real power consumption due to the operation in the transistor level. This attacker would be possible to model behaviorally in a simulations. However, an artificial model for this attacker is not created in the simulations, due to aforementioned reasons.

5.1.4 Fault Injection Attack

In real life it is possible with invasive physical access, by glitching the clock or the voltage to jump to certain branches thus skipping security controls. Similar to the power analysis, it is not built in to the simulation it can be potentially modeled by creating an affect of jumps to force skipping some critical instructions. The modeled fault injection clearly a demonstration of the skip behavior.

5.2 Attack Path Analysis in HSM MMIO Model

The attack path in this architecture is depicted in the Figure 5.3.

Based on the PMP validation experiment explained in using the Algorithm 1 demonstrated the results of this attack paths will be explained in the in the Section 6.

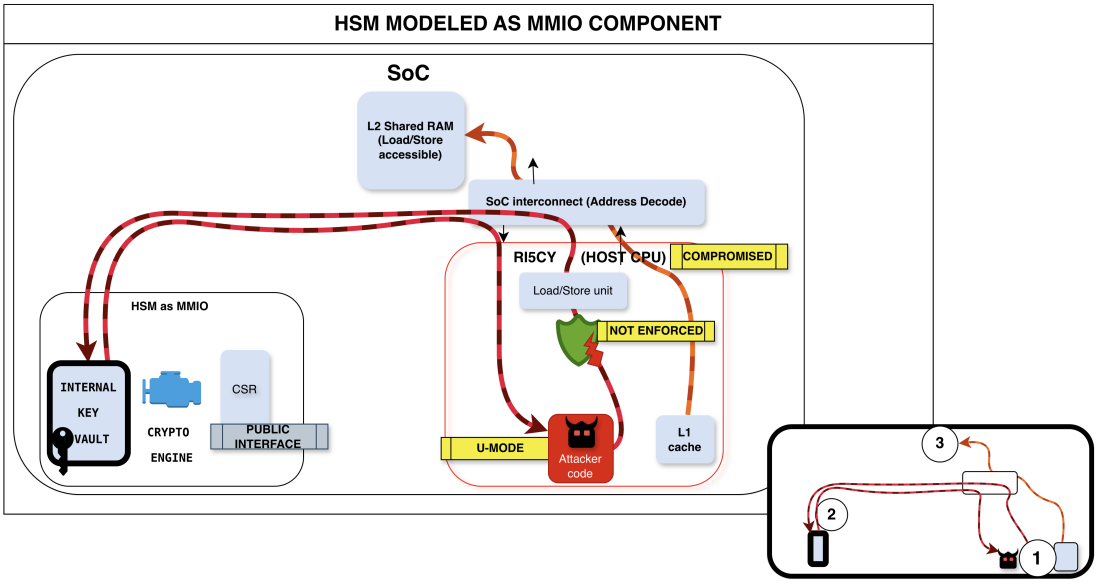

Figure 5.3: Attack path representation for HSM as MMIO model

The attack path, is specifically for the co-resident attacker model and it starts from the attacker code bypassing the PMP protection that was not enforced. As demonstrated the the PMP experiments, in the user mode the load still would succeed. The execution then continue and reach to the SoC interconnect. As a consequence the attacker can access internal key vault directly. And can read the secret key. This fundamentally breaks the HSM requirements, which required to provide isolation and key protection in the as a minimum in the secure boot phase 0 stage. The dash orange path is depicting the further exposure of key in the memory hierarchy after the attack the key can be stored in the L1 cache an can leaked to L2 shared RAM.

5.3 Attack Path Analysis in HSM ISA Extension

Figure 5.4 explains the attack path in the HSM model as ISA extension. It considers the co-resident attack model. The attack starts by repeatedly asking for verification of signed malicious primary boot loader image without the necessity of private key. This puts the hardware emulation in the position of verification oracle. Since the time leak side channel is modeled based on the success or failure status of

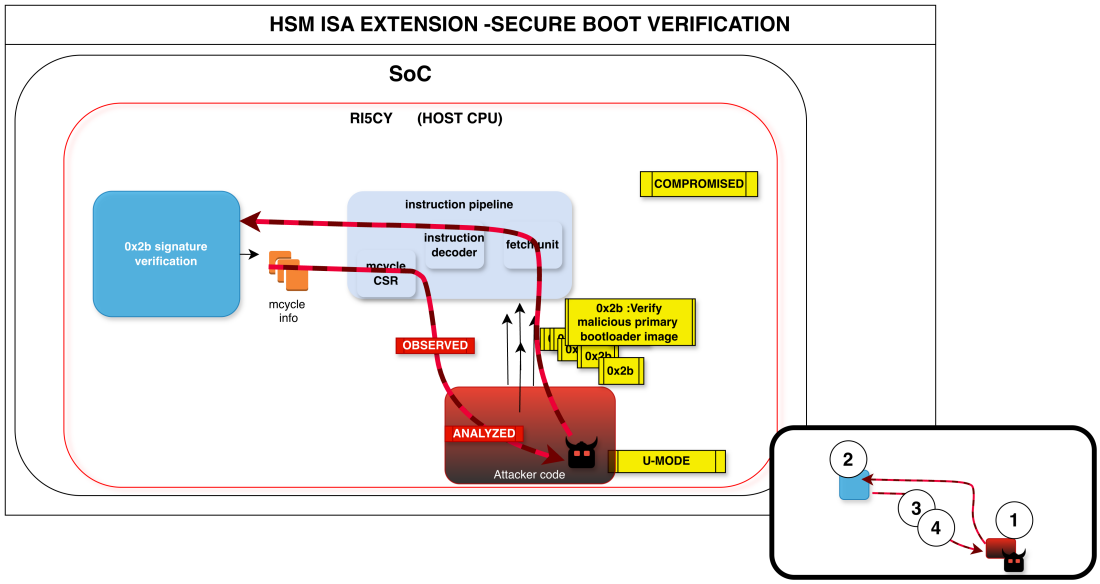

Figure 5.4: Attack path of HSM, modeled as ISA extension

the signature verification this is observable for some attacker models. A *remote attacker* which is outside of SoC can't read return value *rd*. Which means, there is no register level visibility. Nevertheless, they can infer some behavioral pattern from noisier information with respect to a co-resident. In *co-resident* attacker model there is no necessity to observe the time leak in order to understand the result of executing 0x2b. The previous situation holds under the condition that there is no check for the privilege level of who is executing the instruction.

5.4 Security Implications and Consequences

This section will introduce potential consequences that can arise due to implementation of cryptographic operations. It doesn't imply the functionality is wrong. In fact, the security features can be in place however the leakage of information as a result of implementation weaknesses can result as loss of security.

Different versions of timing related side channel attacks exists. In our analysis we want to describe what are attackers capabilities based on the specific scenarios that lead to security loss. In the section 4.5.2, we demonstrated the simple signature verification experiment with the custom opcode and fused key in the instruction handler. During the experiment since the signature verification in place is a simple digest and signature like light mixing scheme, the real cryptographic security is not realized. Rather, it serves only to demonstrate the secure boot control flow and HSM's integration with new ISS. Thus remarking the objective is architectural

analysis, not cryptographic robustness, which is left as a future customization. Nevertheless, an HMAC computation can take place within HSM's computations. Then, it is meaningful to make analysis of attacks and consequences with respect to scenarios i.e., HMAC based integrity protection/ Public Key cryptography for signatures. This will generalize our analysis, since realistically HMAC like methods are not suitable for secure boot verifications)

5.4.1 Attacker Differentiates Success or Failure from Static Time Difference

As it will be presented in more detail in the Figures 6.3 and 6.4 in chapter 7 section 6.2, we demonstrate a side channel model of remote-attacker observing one operation taking longer than the others. It would be possible to understand the oracle like behavior if the total verification time throughout the guesses was always long and only once shorter.

This first observation is not practical for forge of another valid signature byte sequence alone, but still gets information about the operations result. This is posing a danger as explained in this research about the side channel timing attack [25]. It is an important information that can be escalated to a bigger exploration of knowledge if coupled with additional observations. More specifically for proper public key signature verification, pair secret key SK is not stored in the HSM. The attacker therefore can't try to steal the key from HSM, attacker's goal is to forge single signature that Verify(PK, Forged_Signature, Malicious_Image) resulting as SUCCESS. Possible attack is the DoS (Denial of Service) by repeated invocation of *0x2b*, if there is no rate limit.

5.4.2 Byte by Byte Signature Forgery

It is similar to the previous vulnerability due to time based side channel but, this time leak dependents prefix match, i.e., ***strcmp*** early exit on first mismatch as explained in chapter 1 of the work by [8], related to the key or signature verification, then attacker can use the time information not only for deducing the result of verification as success or failure, but also for querying repeatedly the guessed signature changing only one byte at a time. If the implementation is doing an early exit in a wrong signature, attacker can guess byte by byte comparison stop early in first mismatch. Then a valid tag byte naturally will take more time without considering the other timing related delays.

Analysis for the attack complexity considering the presence of implementation weaknesses: For message authentication and data integrity if HMAC is used in computation the brute force complexity to forge a signature is $\mathcal{O}(n \cdot 2^8)$ where n is the tag size. As an example, HMAC-SHA-384 (tag is <= 48 bytes), assuming that

the tag is exactly the size of the digest output, time leak during tag comparison: Without the side channel the brute-force would require 2^{384} try, which is not feasible, however with this information guessing 1 byte at a time the guessed byte sequence has the length 64 bytes. For guessing 1 byte we need 2^8 try as a consequence the whole tag would require $48 \cdot 2^8 = 12288$ attempts computational complexity. A similar analysis vulnerable function can be observed in [26] p.$\tilde{8}$.

Finally it is important to note that even if more strong attacker co-resident considered and byte-level time leak due to early exit, asymmetric cryptography doesn't allow the attacker forging fake signatures. The math involves the ECDSA is not same as HMAC and this point explained more in detailed in the [27] trying to forge a valid signature one byte at a time is not feasible.

5.4.3 Key Re-construction and Multiple Signature Forgery

In order for this attack to be realistically possible the scenarios at hand should be considered. If the digital signature verification is handled with public key cryptography as seen in the diagram 4.18 then there is no secret key stored in the key vault. This is a realistic and correct way as per usual. It is the PK and naturally there is nothing secret about it and it is bound to its private key counterpart. In fact this property is one reason the public key is a HRoT since its change would invalidate the signature. Private key in real scenarios stay in the OEM's backend PKI.

However if the device is using a shared symmetric key and there are previously mentioned weaknesses, and other side channel leakages withing the HMAC, it can result with the attacker recovering the secret key. Which leads to possibility to construct many valid signed data.

Chapter 6

Results

6.1 Strategy 1: Experimental Evaluation of PMP Support in GVSoC

The verification of presence or absence of PMP was a required information for continuing with a specific design strategy. Since the HSM need the properties isolation, key protection and memory isolation based on AUTOSAR documents. As mentioned previously, HSM modeled as MMIO component is meaningful only if there is proper active PMP protection in place. During the experimental runs the behavior of the results in terms of access to the protected memory after the PMP protection was different from the expected one. The Figure 6.1 shows our observation in execution of the MMIO component.

The Figure 6.1 shows initials logs to confirm the setup of MMIO component and immediately continues with PMP experiment results. The observed values are 0x11111111, 0x22222222, 0x55555555, 0x66666666, 0x00000002, 0x00000000. Important flags here are 0x66666666 as it shows the ***load*** actually succeeded for the **(R/W/X)** denied address. Later 0x00000002 proved that the core was truly in the U-Mode. Since the access to the M-mode CSRs are trapped with mcause=2 regardless of the PMP mechanism. Meaning the program stopped due to an illegal instruction.

Based on the observations of this experiment, it is concluded that, PMP enforcement is not effective in this setup. This results led to another strategy as mentioned earlier as strategy 2. The aim of the strategy 2 is to shrink the attack surface to a custom instruction decode instead of accepting the bigger attack surface due to the shared memory and SoC interconnect threats. The analysis of the success of the custom ISA extension logic will be evaluated in its dedicated section.

```
>>> my_system.py imported
>>> Rv32 instantiated
>>> Rv32: binary parameter = /home/ozge_pulpvm/pulp_workspace/gvsoc_clean/theis_project/build/test/test
>>> Rv32: creating Clock_domain and Soc
>>> Rv32: Clock_domain OK
>>> Rv32: creating Soc
>>> Soc instantiated with binary: /home/ozge_pulpvm/pulp_workspace/gvsoc_clean/theis_project/build/test/test
>>> Soc(): creating Router
>>> Soc(): Router OK
>>> Soc(): creating MyComp
>>> Soc(): MyComp created
>>> Soc(): MyComp mapped
>>> Soc(): creating Memory
>>> Soc(): Memory OK
>>> Soc(): mapping Memory
>>> using the RiscvCommon <class 'cpu.iss.riscv.RiscvCommon'>
>>> Soc(): Memory mapped
>>> Soc(): instnatiated RiscV CPUm binary = /home/ozge_pulpvm/pulp_workspace/gvsoc_clean/theis_project/build/test/test
>>> Soc(): CPU ports connected
>>> Soc(): creating ELF loader
>>> Soc(): Loader OK
>>> Soc(): wiring loader signals
>>> Soc(): loader wired
>>> Soc initialization completed
>>> Rv32: Soc OK
>>> Rv32: connecting Clock_domain to Soc
>>> Rv32: clock connected
MyComp mmio write off=0x0 size=4 data=0x11111111
MyComp mmio write off=0x4 size=4 data=0x22222222
MyComp mmio write off=0x0 size=4 data=0x55555555
MyComp mmio write off=0x0 size=4 data=0x66666666
MyComp mmio write off=0x0 size=4 data=0x00000002
MyComp mmio write off=0x4 size=4 data=0x00000000
```

Figure 6.1: mmio experiment result

6.2 Strategy 2: Evaluation of ISA Extension Secure Boot Experiment

```
Launching GVSOC with command:
gvsoc_launcher_debug --config=gvsoc_config.json
Secure boot opcode (0x2b) => PASS
Tampered signature => FAIL
```

Figure 6.2: ISA opcode secure boot phase 0

Figure 6.2 shows the result (PASS/FAIL) of a signature verification after the invocation of the custom opcode 0x2b. The operation will *PASS* for expected valid signature and, *FAIL* due to a controlled tampering to the signature during the experiment. The "*Tampered signature => FAIL*" printed message is due to the logic set in the code section below. The signature is altered in a controlled way by signature 0̂x3u thus the tampered signature should fail and indeed it does as shown in the program print.

```
1 ...
```

```
2     printf("Secure boot opcode (0x2b) => %s\n", verified
      ? "PASS" : "FAIL");
3    uint32_t tampered= my_instr(digest, signature ^0x3u);
      //controled tampering
4     printf("Secure boot opcode (0x2b) => %s\n", tampered
      ? "PASS" : "FAIL");
5     return 0;
6     }
```

6.3 Time Side Channel Modeling

As shown in the Figure 6.2, the secure boot signature verification is invoked by the custom instruction 0x2b. At that stage of the experiment the result of controlled tampering did not involve time modeling or any time side channel vulnerability. In this section by introducing timing model, the Figure 6.3 shows the first example for the time leak. It displays a time difference of 11 cycles between the success and failure. This is not random, it is exactly reflecting our modeling of a vulnerability. Note that with this model the time leak is not due to a dependence of early exit at byte level. A byte level early exit would not be constant time difference, as the compared hash would be different per signature verification. That would be considered as a bigger leak of information.

Since the experiment to show this vulnerability is adding additional time for the failed operation, the co-resident attacker can observe the *mcycle* to deduce the operation success or failure from time based side channel. However, this observation is not relevant information for the co-resident attacker, since if the malicious software is running in the CPU it already can access to the experiment result. Thus the time leak at this point is relevant (although it is weaker) to the remote attacker that is outside of SoC.

```
Launching GVSOC with command:
gvsoc_launcher_debug --config=gvsoc_config.json
--- ATTACKER SIMULATION START ---
Guess [0xdeadc0de] Result: 0 | Time: 21 cycles
Guess [0xd49818d2] Result: 1 | Time: 10 cycles

[!] SIDE-CHANNEL ALERT: Timing difference detected!
The HSM is leaking information. Difference: 11 cycles.
Logic: Faster execution indicates NO STALL => Signature is CORRECT.
```

Figure 6.3: ISS opcode for HSM secure boot verification

Similar to the results observed in the Figure 6.3 the Figure 6.4 is demonstrating the side channel vulnerability but this time the computation of the signature verification has the total cycles more realistic time referenced from the selected white paper [21].

```
Launching GVSOC with command:
gvsoc_launcher_debug --config=gvsoc_config.json
--- ATTACKER SIMULATION START ---
Guess [0xdeadc0de] Result: 0 | Time: 2729 cycles
Guess [0xd49818d2] Result: 1 | Time: 2718 cycles

[!] SIDE-CHANNEL ALERT: Timing difference detected!
The HSM is leaking information. Difference: 11 cycles.
Logic: Faster execution indicates NO STALL => Signature is CORRECT.
```

Figure 6.4: HSM opcode with time side channel modeling

6.4 Fault Injection Modeling

Fault injection is a physically invasive attack that requires externally applied perturbations to the target device, by various methods including voltage glitching, clock manipulation, or electromagnetic pulses. This techniques could lead to bypass security critical execution paths. By causing instruction jump to another leading to skip of the security check.

In order to model the physical fault injection behavior in a controlled environment. It is the *behavior model* because clearly it is not realistic to simulate the physical attack in a non existing board. GVSoC can be used to demonstrate glitched security bypass in a symptom level simulation.

This modeling is meaningful because it explores potential weakness. Specifically the conditions under which a fault can bypass a security-critical check. Simulation allows testing the resiliency of the custom opcode against fault injection attacks bu reproducible and accessible foundation for the evaluations.

```
Launching GVSOC with command:
gvsoc_launcher_debug --config=gvsoc_config.json
--- ATTACKER SIMULATION START ---
[FAULT] my_instr verification skipped
[FAULT] my_instr verification skipped
Guess [0xdeadc0de] Result: 1 | Time: 11 cycles
Guess [0xd49818d2] Result: 1 | Time: 10 cycles

[!] SIDE-CHANNEL ALERT: Timing difference detected!
The HSM is leaking information. Difference: 1 cycles.
Logic: Faster execution indicates NO STALL => Signature is CORRECT.
(~pulp_venv) ozge_pulpvm@pulpvm:~/pulp_workspace/gvsoc_clean/core/models/devices/tutorial_hsm$ printenv GVSOC_FAULT_SKIP_MYINSTR
1
```

Figure 6.5: Experiment result of fault injection model `activated`

Figure 6.5 demonstrates the result obtained by activating the fault injection behavior setting the **GVSOC_FAULT_SKIP_MYINSTR** environment variable to 1. After this the 2 prints of custom instruction skipped is printed since the last experiment involves call to 0x2b with valid and altered signature. A more interesting result is observed in the **Result** value. As the correct signature should have the result value 1, this time also the modified signature leads to the correct value. In comparison to the results demonstrated in the Figure 6.6, no fault injection scenario the environment variable set to 0. Consequently the signature verification operation takes place and crucially the time model with artificial cycle time this time is 2729 and 2718 cycles. As observed in the Figure 6.5 the bypassed signature verification is taking only 10/11 seconds.

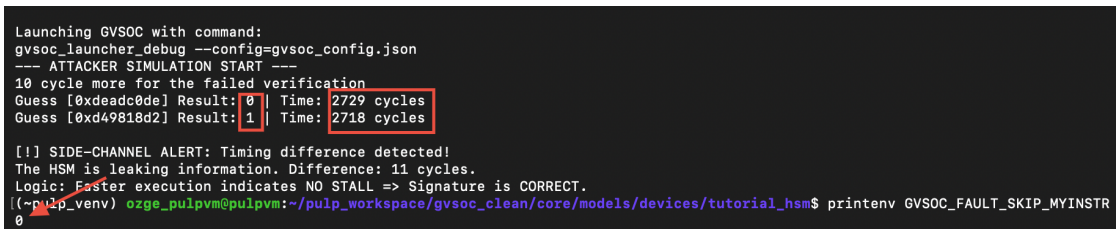

Figure 6.6: Experiment result of fault injection model `not activated`

6.5 Limitations and Challenges

Since this study aims to have a security critical component in a simulation environment without a physical real HRoT faced couple of challenges are present.

1. To realistically emulate an immutable Boot ROM and HRoT behavior the absence of real OTP or eFuse like mechanism to store a public key doesn't exist in the simulation world. Thus the HRoT in the process level can be emulated but in physical component level is not possible to emulate.

2. Realistic threat modeling for some of the well known threats are not meaningful in the simulation environment. I.e., DPA/CPA. Moreover, result of glitch behavior can be modeled, but it is limited in reflecting a realistic attack of a fault injection attack.

3. Most of the state of the art have HSM model in an isolated separate component, this wanted to be achieved in open source with MMIO component model as our first strategy. It is assumed that the PMP is a present protection in the selected core document [28], however even after the several activation attempts it was not enforced. The root cause for this behavior was confirmed

in the concluding stages of the research upon the discovery of supplementary architectural manuals from the [29] OpenHW group. According to CV32E40P user manual "CV32E40P core does not support the RV32A (atomic) extensions, the U-mode, and the PMP **anymore**."

4. The implementation of custom signature verification instruction is linked to the instruction fetch and decode withing a simulation environment. This abstraction of the real world means the functional accuracy depends of simulator's micro architectural representation of the cycles.

Strategy	Security Property	Security Properties Missing	How similar to a real HSM (key isolation+controlled interface)	is/not implemented ? Why ?
HSM modeled as MMIO component with PMP enforcement	Hardware enforced memory isolation between M-mode / U-mode	Partially isolated and protected key vault, still necessary to model bus firewall agains the DMA bus masters not a physical real isolation	PMP could only provide software level access policy to a protected memory region.	NO-> since experiment to check the PMP protection is concluded missing enforcement
Only ISA extension for secure boot verifcation	Controlled entry point software can only request an operation through customer instruction it conceptually hide implementation behind the an interface emulation of accelerator style service call	No isolation of HSM from ri5cy as as separate component. No secure interface(regular load/store instruction) , Key is not protected with physical protected key vault.	No addressability by co-resident attack model. Software can not load the key via normal memory address. It can only ask for HSM verification service. The key is not in ROM/RAM. It is in the	YES-> ISA extension is supported in GVSoC and was significantly reducing the key protection+isolation in the absence of PMP.
PMP+ ISA extension	PMP isolate called and data in the specific protected region ISA custom of code provides minimal path to use the HSM service	Key is not protected with physical protected key vault.	Code and data(key) in the HSM's address space is protected against co-resident attackers read/write. Minimal verification opcode path is hard to abuse	NO and no real advantage in the key protection without a bus firewall

Figure 6.7: Evaluation summary table

6.6 Evaluating Validity of Simulated Hardware Isolation for the Key

A critical concept that is evaluated is the feasibility of achieving isolation and protection of the hardware trust anchor with sensitive material i.e., keys. In physical HSM's HRoT is achieved by OTP/eFuse like permanent storage. Moreover there should have been the equivalent of hardware enforced memory isolation for the access protection.

Knowing the limitation for the lack of OTP/eFuse like permanent storage of the key and no physical memory protection in our chosen core and simulation, memory mapped key storage encapsulated in an isolated HSM component was not feasible. Instead, direct access implementation removed key's presence from architectural shared path. It is done by injecting the key to the custom instruction handler itself. As such the security properties of the physical world is not available however,

architectural isolation and secure boot flow can demonstrate keys were never in the systems memory map.

6.7 Evaluating validity of simulated process protection

The chosen design methodology as custom opcode to offload the signature verification not only avoids lack of proper protection due to the PMP enforcement, it also provides a deeper layer of security in process execution level. HSM's secure boot signature verification operation is offloaded to a single opcode and all the execution is forced to start from a carved ROM memory region. Against the natural limitation configuration in place SoC emulation demonstrates the start from Boot Rom achieving architectural and process level isolation.

Chapter 7

Future Work & Research Directions

The secure boot architectural flow developed in this study, particularly the custom code interface and the ISA extension strategy is specifically designed for re-usability and extensibility. An important future direction there for would be the substitution of the current, simplified mixing function with a fully asymmetric primitives and further with post-quantum cryptographic algorithms recently standardized by NIST. The open-source and custom instructions showcase flexibility that's all architectural components from the custom opcode definition to the platform integration. They are designed to be the replaceable in isolation. It does this by allowing new features to swap or add new cryptographic primitives without restructuring, the surrounding security architecture.

Additionally , the platform independence nature of the RISC-V increases the reliability making the architectural flow suited for evaluation across heterogeneous target chips, a property of direct relevance to automotive security, where ECUs from different vendors must satisfied common security requirements following ISO 21434 standards.

Chapter 8

Conclusion

Modern automotive systems increasingly rely on broad set of security guarantees that require process and component level trust anchor. A component that is performing efficient cryptographic operations and verifications during firmware updates. This work studied a foundational architectural model of HSM, a component that is extensively used for critical cybersecurity functionality. This thesis demonstrated secure boot flow as selected HSM functionality as it is a critical chain of verification and origin of security for the rest of the ECU lifecycle. The design implemented in a customizable and open-source PULP platform within a cycle accurate GVSoC simulation environment. Demonstrated experiments focused to close the gap between extensibility requirements of security functionality and black-box proprietary solutions. It prioritized transparent design that would allow full threat model analysis of this and future research applications using open-source PULP platform.

Two architectural strategies explored to achieve the isolation property. The first treated the HSM as protected peripheral, reflecting a well-known design strategy. While it was conceptually aligned and easier to implement as a component model, a missing enforcement of physical memory protection was found to be the practical limitation in the RI5CY core. This result deduced with experiments and later solidified with a document discovered retroactively.

The second strategy implemented early stages of secure boot verification as a custom instruction to demonstrate one of the most important HSM functionality. The custom HSM opcode created as an architectural extension with the RISC-V instructions set simulator. Through embedding cryptographic verification inside an instruction level primitive, this method ensured that no sensitive material is present in the shared address space. Ultimately achieving much smaller attack-surface with respect to protected-peripheral model of HSM. This approach more closely resembles accelerated instructions, that gives high performance and secure executions. It makes the design more aligned with the acceleration property of the

HSMs within the SoC.

Open-source nature of PULP platform allowed architectural transparency, and deep customization capabilities.
Particularly the support for the custom ISA extensions allowed the creation of signature verification opcode invoked in the secure boot phase 0. GVSoC simulator complemented the features of PULP and allowed validation of the control flow and cycle accuracy and ROM resident immutable entry point. The remaining parts of this thesis presented the threat modeling and attack path analysis. Results of experiments conducted a demonstrates comparative analysis for security posture of two architectural designs belonging to simulated environment under a co-resident attacker model with respect to real HSMs in physical SoC. Finally, two attack scenarios modeled in the simulation environment; time side channel analysis and, fault injections are demonstrated. The threat modeling is constructed based on the attack path analysis, attack models and capabilities.

This thesis positioned as a reference point for future research providing reusable architectural framework and threat modeling. Choosing the tightly coupled HSM strategy and eliminating weak design directions based on experimental results are direct example for future research. The researchers in this field can extend strategies and integrate Post-Quantum algorithms with preferred strategy based on their platform specifications in future applications.

Appendix A

Program & Commands

A.1 Starting the simulation environment

The simulation starts with following commands and after the changes the rebuild and run is repated.

```
source ~pulp_venv/bin/activate
cd pulp_workspace/pulp-sdk/
source configs/pulp-open.sh
cd ~/pulp_workspace/test_working/fork
make clean all run
~/pulp_workspace/gvsoc_clean
source sourceme.sh
make clean
make all
make gvsoc
make run
```

gvsoc/core/models/cpu/iss/include/isa/rv32i.hpp simple iss handler added to the library

```
static inline uint32_t mac(uint32_t digest){
const uint32_t fused_key = 0xa5c3f18du;
uint32_t mac = digest ^ fused_key;
mac = (mac << 7) | (mac >> 32 -7);
return mac ^ 0x3c6ef372u;
}
```

```
7     ...
8     const uint32_t digest = (const uint32_t)REG_GET(0);
9     const uint32_t provided_signature = (const uint32_t)
    REG_GET(1);
10    const uint32_t expected_signature = my_instr_mac(
    digest);
11    const uint32_t verified = expected_signature ==
    provided_signature;
12    REG_SET(0, verified);
```

gvsoc/core/models/cpu/iss/include/isa/rv32i.hpp time leak and realistic time modeled

```
1     static inline uint32_t mac(uint32_t digest){
2     const uint32_t fused_key = 0xa5c3f18du;
3     uint32_t mac = digest ^ fused_key;
4     mac = (mac << 7) | (mac >> 32 -7);
5     return mac ^ 0x3c6ef372u;
6     }
7     ...
8     const uint32_t digest = (const uint32_t)REG_GET(0);
9     const uint32_t provided_signature = (const uint32_t)
    REG_GET(1);
10    const uint32_t expected_signature = my_instr_mac(
    digest);
11    const uint32_t verified = expected_signature ==
    provided_signature;
12    REG_SET(0, verified);
13    iss->timing.stall_cycles_account(2708);
14    if(!verified){
15        iss->timing.stall_cycles_account(2708);
16    }
17    return iss_insn_next(iss, insn, pc);
```

```
(~pulp_venv) ozge_pulpvm@pulpvm:~/pulp_workspace/gvsoc_clean$ grep -R "CONFIG_GVSOC_ISS_PMP" -n install | head
install/generators/cpu/iss/riscv.py:296:                '-DCONFIG_GVSOC_ISS_PMP=1',
install/generators/cpu/iss/riscv.py:297:                '-DCONFIG_GVSOC_ISS_PMP_NB_ENTRIES=16'])
install/generators/cpu/iss/iss.py:128:                '-DCONFIG_GVSOC_ISS_PMP=1',
install/generators/cpu/iss/iss.py:129:                '-DCONFIG_GVSOC_ISS_PMP_NB_ENTRIES=16'])
```

Figure A.1: PMP flags investigation

Figure A.2: PMP functions existence verification

Figure A.3: PMP configuration verification

gvsoc/core/models/cpu/iss/include/isa/rv32i.hpp modification to model the Fault Injection.

```
1  static inline bool fault_skip_now(){
2    const char *v = std::getenv("
   GVSOC_FAULT_SKIP_MYINSTR");
3    if(v == nullptr) return false;
4
5    if(std::strcmp(v, "1") == 0) return true;
6    return false;
7  }
8 static inline uint32_t my_instr_exec(Iss *iss,
    iss_insn_t  *insn, iss_reg_t pc){
9   if(fault_skip_now()){
10       std::print("[FAULT] my_instr verification skipped
   ");
11       REG_SET(0, 1);
12       return iss_insn_next(iss, insn, pc);
13   }
14   ....
15 }
16
```

```
17
18  }
```

Bibliography

[1] AUTOSAR. *AUTOSAR Layered Software Architecture*. Accessed online. URL: https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf (cit. on p. 1).

[2] Wikipedia. *Automotive Safety Integrity Level*. Accessed online. URL: https://en.wikipedia.org/wiki/Automotive_Safety_Integrity_Level (cit. on p. 2).

[3] 2021. URL: https://www.infineon.com/assets/row/public/documents/30/49/infineon-sls37-v2x-hsm-datasheet-en.pdf (cit. on p. 4).

[4] NXP. Accessed online. 2019. URL: https://www.nxp.jp/docs/en/training-reference-material/NEXT-GENERATION-AUTOMOTIVE-SECURITY-SOLUTIONS.pdf (cit. on p. 4).

[5] PULP platform. *Pulp Faqs*. URL: https://pulp-platform.org/ (cit. on p. 4).

[6] Nazareno Bruschi, Germain Haugou, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. «GVSoC: A Highly Configurable, Fast and Accurate Full-Platform Simulator for RISC-V based IoT Processors». In: *2021 IEEE 39th International Conference on Computer Design (ICCD)*. 2021, pp. 409–416. DOI: 10.1109/ICCD53106.2021.00071 (cit. on pp. 4, 16).

[7] Ivan Studnia, Vincent Nicomette, Eric Alata, Yves Deswarte, Mohamed Kaâniche, and Youssef Laarouchi. «Survey on security threats and protection mechanisms in embedded automotive networks». In: *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*. 2013, pp. 1–12. DOI: 10.1109/DSNW.2013.6615528 (cit. on p. 9).

[8] Jasper van Woudenberg and Colin O'Flynn. *The hardware hacking handbook: Breaking embedded security with hardware attacks*. No Starch Press, 2022 (cit. on pp. 10, 38, 47).

[9] Fabian Schwarz, Jan Philipp Thoma, Christian Rossow, and Tim Güneysu. «KeyVisor – A Lightweight ISA Extension for Protected Key Handles with CPU-enforced Usage Policies». In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2025.3 (June 2025), pp. 1–31. DOI: 10.46586/tches.v2025.i3.1-31. URL: https://tches.iacr.org/index.php/TCHES/article/view/12208 (cit. on p. 14).

[10] openHW group. Accessed online. 2023. URL: https://docs.openhwgroup.org/projects/cv32e40p-user-manual/en/cv32e40p_v1.8.3/intro.html (cit. on pp. 14, 16).

[11] PULP. Accessed online. 2019. URL: https://osda.gitlab.io/19/rossi-slides.pdf (cit. on p. 16).

[12] URL: https://openhwfoundation.org/ (cit. on p. 16).

[13] ETHZurich. *GVSoC: A Highly Configurable, Fast and Accurate Full-Platform Simulator for RISC-V based IoT Processors*. Accessed online. 2023. URL: https://arxiv.org/pdf/2201.08166 (cit. on p. 16).

[14] toolchain. Accessed online. 2023. URL: https://github.com/pulp-platform/pulp-riscv-gnu-toolchain/blob/master/README.md (cit. on p. 16).

[15] ETHZurich. Accessed online. 2023. URL: https://pulp-platform.org/docs/riscv_workshop_zurich/schiavone_wosh2019_tutorial.pdf (cit. on p. 17).

[16] URL: https://docs.riscv.org/reference/isa/v1.10/_attachments/riscv-privileged.pdf (cit. on p. 22).

[17] RISC-V International. *The RISC-V Instruction Set Manual: Volume II*. Accessed online. 2025. URL: https://docs.riscv.org/reference/isa/_attachments/riscv-privileged.pdf (cit. on p. 23).

[18] Germain Haugou. *Gvsoc Configuration*. Accessed online. 2023. URL: https://github.com/pulp-platform/gvsoc/blob/master/docs/configuration.rst (cit. on p. 26).

[19] RISC-V. *Immediate Encoding Variants*. Accessed online. 2025. URL: https://riscv.github.io/riscv-isa-manual/snapshot/unprivileged/#base_instr (cit. on p. 27).

[20] Ali Taqweem. *A Beginners Guide to Automotive Bootloaders — Understanding basics*. Accessed online. 2022. URL: https://medium.com/maanz-ai/a-beginners-guide-automotive-bootloaders-understanding-basics-98ce70675206 (cit. on p. 29).

[21] Görkem Nişancı, Paul Flikkema, and Tolga Yalcin. «Symmetric Cryptography on RISC-V: Performance Evaluation of Standardized Algorithms». In: *Cryptography* 6 (Aug. 2022), p. 41. DOI: 10.3390/cryptography6030041 (cit. on pp. 40, 41, 52).

[22] NXP. *Secure JTAG*. Accessed online. 2025. URL: https://docs.nxp.com/bundle/AN4686/page/topics/secure_jtag.html (cit. on p. 43).

[23] ARM. *Securing the debug interface of your devices*. Accessed online. 2022. URL: https://developer.arm.com/documentation/107745/0100/Overview (cit. on p. 43).

[24] PULP. Accessed online. 2023. URL: https://pulp-platform.org/docs/lugano2023/gvsoc_pulp_anniversary.pdf (cit. on p. 44).

[25] Paul C. Kocher. «Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems». In: *Advances in Cryptology — CRYPTO '96*. Ed. by Neal Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113. ISBN: 978-3-540-68697-2 (cit. on p. 47).

[26] Dr. Ing. M. Heiderich, Dr. D. Bleichenbacher, and Dr. S. Mazaheri. *Audit-report NIP44 implementations 11.-12.2023*. Dec. 2013. URL: https://cure53.de/audit-report_nip44-implementations.pdf (cit. on p. 48).

[27] Soatok. *Dead Ends in Cryptanalysis 2: Timing Side-Channels*. Accessed online. 2021. URL: https://soatok.blog/2021/06/07/dead-ends-in-cryptanalysis-2-timing-side-channels/ (cit. on p. 48).

[28] 2019. URL: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf (cit. on p. 53).

[29] OpenHW. *ri5cy*. Accessed online. 2023. URL: CORE-V%20CV32E40P%20User%20ManualOpenHW%20Foundationhttps://docs.openhwgroup.org%20%E2%80%BA%20downloads%20%E2%80%BA%20epub (cit. on p. 54).