UNIVERSITY OF TRENTO

Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

Final Dissertation

Design and Implementation of Trusted Services on RISC-V: the case of Control Flow Integrity

A lightweight Control Flow Integrity Enforcer for embedded RISC-V systems

Supervisor
Bruno Crispo
Co-Supervisor
Michele Grisafi

Student
Davide Moletta
239272

Academic year 2023/2024

Contents

Abstract

The proliferation of embedded devices across industries and daily life, from consumer electronics to healthcare and industrial automation, has amplified the urgency of securing these systems against an escalating array of cyber threats. This thesis presents a novel security framework designed to enforce Control Flow Integrity to protect embedded systems based on the open *RISC-V* Instruction Set Architecture. Unlike proprietary ISAs, *RISC-V* enables the development of customized processors and solutions tailored to specific use cases. This project exploits this flexibility to address control flow hijacking attacks, such as *Return-Oriented Programming* and *Jump-Oriented Programming*, which threaten the integrity of execution paths in microcontroller-based devices. Such attacks allow attackers to gain control over the target device, execute arbitrary code, or exfiltrate sensitive data, making them a significant threat to modern-day embedded devices.

The proposed solution incorporates lightweight Control Flow Integrity enforcement mechanisms, including a shadow stack and Control Flow Graph validation, to ensure that software execution follows predefined paths. This prevents unauthorized deviations caused by potential attackers, ensuring the security of the system and its data. The solution employs code instrumentation techniques that automatically integrate Control Flow Integrity into user applications, simplifying deployment and reducing development time. Moreover, using *RISC-V*'s Physical Memory Protection we can safeguard critical data structures, enhancing the framework's robustness.

The design philosophy behind the project emphasizes efficiency and scalability, enabling it to operate effectively on resource-constrained embedded devices. A key feature is its ability to provide strong security guarantees with minimal performance overhead, making it suitable for IoT devices that often have limited computational and power resources. A Proof of Concept implementation demonstrates the effectiveness of our solution in defending against control flow hijacking attacks. Detailed performance analysis reveals that the Control Flow Integrity enforcer achieves a balance between security and operational efficiency, with an evaluation of memory overhead and execution time, supporting its feasibility for practical applications.

Furthermore, this thesis explores the integration of Real-Time Operating Systems such as *FreeRTOS* and *Zephyr RTOS*, showcasing the project's adaptability to complex software environments. This integration broadens the applicability, enabling the secure execution of real-time and multitasking applications in IoT and embedded domains.

In conclusion, the proposed solution addresses the pressing challenge of securing embedded *RISC-V* devices by combining innovative security mechanisms, automated processes, and adherence to *RISC-V*'s open and customizable principles. This work aim to contribute significantly to the field of embedded system security by offering a scalable, efficient, and customizable framework that meets the needs of modern IoT environments while anticipating future challenges in the cybersecurity landscape.

1 Introduction

This chapter serves as an introduction to this thesis, where we present a new security infrastructure for *RISC-V*-based embedded devices. The proposed solution aims to provide security features to mitigate control flow hijacking attacks and allow the execution of untrusted user code. This result is accomplished by enforcing Control Flow Integrity on the target code. Moreover, great focus has been put on providing an efficient and lightweight infrastructure that complies with the limited hardware capabilities of embedded devices.

1.1 Problem

Embedded devices are now an integral part of daily life, powering a wide range of applications, from smart home assistants and appliances to connected furniture. These compact computing units have expanded across numerous domains, supporting diverse tasks that enhance convenience, efficiency, and connectivity in everyday environments. A defining feature of embedded devices is their network connectivity, which links them to the broader Internet of Things (IoT) ecosystem. According to the "*State of IoT Summer 2024*" report, 16.6 billion devices were connected by the end of 2023, a figure expected to rise to 18.8 billion during 2024[1]. These devices perform critical functions across industries ranging from consumer electronics to industrial automation and healthcare. However, they are often resource-constrained and designed to perform specific tasks with minimal power and computational resources, making them inherently vulnerable to cyber threats. In 2022, approximately 112 million IoT cyberattacks were reported globally[2], and this figure continues to escalate, with the first half of 2024 witnessing a 107% increase in attacks as highlighted by *SonicWall*'s "*2024 Mid-Year Cyber Threat Report*"[3]. Consequently, embedded device security is an increasingly pressing concern, as attacks on these devices can result in data breaches, unauthorized system control, or disruptions to essential services.

Among the potential threats, we have software-based, network-based, and side-based attacks that an attacker may use to impact the Confidentiality, Integrity, and Availability of a system. One notable example of a network-based attack is the *Man-In-The-Middle* (MITM) attack. In this scenario, a malicious actor intercepts the communication between two parties, thereby gaining unauthorized access to sensitive data such as cryptographic keys, login credentials, and personal information. This breach can lead to severe consequences, including data theft, identity fraud, and loss of trust in communication channels. Another example is the threat posed by control flow hijacking attacks, a family of cyber attacks that aim at disrupting the normal execution flow of a program to execute arbitrary code or gain control of the target device. These types of attacks may result in additional threats, for example, if an attacker manages to gain control of a significant number of devices, they can form what is known as a *Botnet*. This network of infected devices can be orchestrated to launch large-scale *Distributed Denial of Service* (DDoS) attacks against external systems. In a DDoS attack, the botnet floods a targeted server or network with excessive traffic, overwhelming it and rendering it unable to process legitimate requests. This disruption can lead to service outages, financial losses, and damage to the organization's reputation.

With all this information it is easy to understand why embedded devices have become a suitable target for cyber attacks and why this is a major security concern. It would be relatively "easy" to think of implementing state-of-the-art security solutions to protect these devices, however, the problem is that many of them have limited hardware resources as they may be designed to carry out simple tasks. This means we must apply security measures within the device's capabilities without affecting

[1] `https://iot-analytics.com/number-connected-iot-devices/`

[2] `https://1nce.com/en-us/resources/news/blog/iot-cybersecurity-landscape`

[3] `https://www.sonicwall.com/threat-report`

performance. This is a challenge because, in embedded systems security, we often need to design tailored security features to protect the device while maintaining high efficiency. In this thesis, we discuss how we aim to contribute to the world of embedded security with the presented project.

1.2 Goals

This project introduces innovative security features designed specifically for embedded *RISC-V*-based systems. The primary objective of this project is to establish a robust software-based Control Flow Integrity enforcer that allows secure execution of untrusted user code. This is particularly crucial in scenarios where running potentially malicious or unverified code can pose significant risks to system integrity and security. A key focus of the project is to ensure that Control Flow Integrity remains intact throughout the execution process, thereby preventing any unauthorized modifications to the flow of control within the program. This protection is achieved by enforcing backward and forward edge protection using state-of-the-art techniques such as a shadow stack and Control Flow Graph validation. Moreover, Physical Memory Protection is integrated into the project to protect critical memory areas.

Additionally, considering that this project aims at protecting embedded devices, which often have constrained hardware resources, special emphasis has been placed on employing optimization techniques. These techniques are designed to minimize the impact of the Control Flow Integrity enforcer on both memory consumption and execution performance. We aim to enable the secure execution of untrusted code without imposing excessive overhead that could hinder the functionality of resource-limited embedded systems.

Furthermore, throughout the development process, we have adhered to the open and modular design principles inspired by *RISC-V*. This commitment to modularity and openness has resulted in a highly adaptable infrastructure that can be easily modified to suit a variety of deployment scenarios. With this, we aim to streamline the deployment process, making it faster and more efficient in diverse environments, thereby enhancing the overall utility and effectiveness of provided security features.

By integrating Control Flow Integrity mechanisms, this project aims to enhance the security of control paths within *RISC-V* systems. Control Flow Integrity ensures that the execution of the embedded system remains trusted and reliable, safeguarding it from potential exploits or vulnerabilities that could compromise its operations. Through this advancement, the project not only improves the robustness of the *RISC-V* instruction set architecture but also contributes to the overall resilience of embedded applications in various critical fields.

1.3 Structure

Following this introduction, chapter 2 presents a comprehensive background about the problem, security techniques, and related works that provide a similar solution to the one presented in this thesis. We start with the problem's description, listing the relevant attacks we aim to mitigate. Secondly, we discuss state-of-the-art security measures against control flow hijacking attacks, strongly focusing on Control Flow Integrity. Lastly, we present related projects discussing what differentiates them from our approach.

In chapter 3 we provide the threat model and assumptions for this project. We state the problem and the target of our security measures. Also, we discuss assumptions we made about the threat scenario and the capabilities of an attacker.

Chapter 4 presents the foundational concepts of the *RISC-V* Instruction Set Architecture. In this chapter, we focus only on essential information needed for a complete understanding of the architectural basis of the project.

Chapter 5 details the implementation specifics of the Control Flow Integrity enforcer, discussing critical features such as forward and backward edge control mechanisms and the use of Physical Memory Protection to create secure memory spaces. To validate the effectiveness of the project in real-world scenarios, this chapter also introduces a Proof of Concept which will exemplify how the project addresses critical security threats, illustrating its impact on device resilience and system integrity.

A subsequent security analysis, discussed in chapter 6, will describe the testing methodologies used to assess the project's effectiveness in identifying and preventing control flow hijacking attacks. Moreover, we discuss the security limitations of the Control Flow Integrity enforcer identifying any potential vulnerabilities that may need further consideration.

A performance analysis is provided in chapter 7, where the trade-offs between security and system efficiency are evaluated. This analysis will help highlight the strengths and limitations of the project and explain why it represents a feasible solution for enhancing the security of *RISC-V*-based embedded devices. During this chapter we showcase test results for time and memory consumption, depicting the average overhead of the project. Lastly, we provide insights on optimization ideas to further enhance performance.

Additionally, the thesis explores the potential integration of a Real-Time Operating System (RTOS) during chapter 8. In particular, we propose the integration of *FreeRTOS* and *Zephyr RTOS*, two widely-used Real-Time Operating Systems. Such integration could extend the project's applicability, enabling more complex programs to operate under the Control Flow Integrity enforcer. By supporting an established RTOS like *FreeRTOS*, the project could broaden its reach and utility, making it suitable for a larger variety of use cases within the embedded device domain.

Finally, in chapter 9, we discuss future enhancements we plan to integrate. In this chapter, we mainly focus on security and performance-oriented solutions that could broaden the security of the project while reducing its impact on the device's performance.

In summary, this thesis presents a novel approach to securing *RISC-V* microcontrollers, leveraging Control Flow Integrity to protect against control flow hijacking attacks and providing a secure foundation for running potentially unsafe code. Through detailed technical exploration, demonstration, and analysis, this work aims to contribute a significant advancement to the field of embedded device security, addressing both current and emerging threats in the IoT landscape.

2 Background

The purpose of this chapter is to provide a comprehensive description of the attacks that we aim to mitigate with the implemented security features. We start by listing the main threats and discussing how they can disrupt the execution flow of the system to cause damage. Secondly, we describe possible solutions to these types of threats focusing on Control Flow Integrity, its purpose, and how this security measure can effectively detect and prevent control flow hijacking attacks. Lastly, we present related works discussing the differences between them and the novel solution proposed in this thesis.

2.1 Control Flow Hijacking Attacks

Control flow hijacking attacks are a category of cyber attacks that aim at disrupting the normal execution flow of a program. Normally, some code follows one or more paths during execution, control flow attacks aim at modifying such flow by transferring control to existing or injected malicious code. The purpose of these types of attacks is to alter the program's flow to leak information, gain higher privileges, execute arbitrary code, or, in general, hijack the system.

This project is focused primarily on detecting and preventing control flow hijacking attacks on *RISC-V*-based embedded systems. Here, we provide a list of these types of threats showcasing how they can affect the execution path to impact the system.

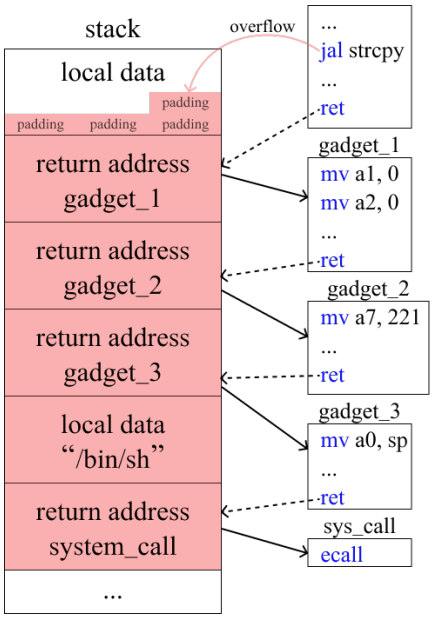
2.1.1 Code Reuse Attacks

Code Reuse Attacks are a class of exploit techniques in which an attacker uses existing, legitimate code within a program to execute malicious actions, circumventing protections like non-executable memory regions. These attacks exploit the fact that many software protections assume that code residing in executable regions of memory is safe, and they focus instead on preventing the execution of data in memory. By leveraging already present and authorized code snippets, attackers avoid introducing new code, making detection and prevention more challenging.

One common example of a *Code Reuse Attack* is *Return-Oriented Programming* (ROP). In ROP, an attacker utilizes a sequence of "gadgets", which are small chunks of code ending in a return instruction. These gadgets are typically harvested from the binary or shared libraries of the targeted program and perform specific operations. By chaining gadgets together through careful manipulation of the program's stack, the attacker creates a sequence of instructions that accomplish a broader malicious goal, such as disabling security mechanisms, executing shellcode, or gaining unauthorized access. Figure 2.1 depicts the principle of *Return-Oriented Programming*.

Jump-Oriented Programming (JOP) is a variation of ROP, where instead of relying on return instructions, attackers rely on jump or call instructions to chain gadgets. Similarly to ROP, these approaches exploit the control flow of the application without introducing new code into the memory space.

Code Reuse Attacks are particularly dangerous because they bypass certain forms of security, such as signature-based detection mechanisms and runtime protections that rely on identifying or blocking the introduction of foreign code. Techniques like Address Space Layout Randomization attempt to mitigate these attacks by randomizing the locations of code in memory, but attackers may combine information leakage vulnerabilities



Source: *Return-Oriented Programming* on *RISC-V* [11]

Figure 2.1: *Return-Oriented Programming* principle

or brute force techniques to bypass Address Space Layout Randomization and successfully locate the needed gadgets.

The sophistication of *Code Reuse Attacks* lies in their ability to manipulate a program's legitimate functionality to achieve unintended behavior while leveraging the program's permissions and resources. This makes them a preferred method for attackers seeking reliability in their exploits.

2.1.2 Stack Smashing Attacks

A *Stack Smashing Attack* is a type of exploit that targets vulnerabilities in a program's use of memory, specifically the stack, to gain unauthorized access or control over the program's execution. It typically occurs in applications that improperly handle input data, allowing attackers to overwrite parts of the stack with their own crafted data.

The attack begins with exploiting a buffer overflow vulnerability, which arises when a program allocates a fixed-size buffer on the stack to store user input but fails to check whether the input exceeds the buffer's size. When the attacker provides input that surpasses this limit, the excess data spills over into adjacent memory regions of the stack. These regions might contain critical information, such as the function's saved return address or local variables.

By carefully crafting the overflowing input, an attacker can overwrite the saved return address with the address of malicious code previously injected into the memory or a code reuse sequence. When the function attempts to return, it jumps to the attacker-controlled location instead of the intended one. This redirection allows the attacker to execute arbitrary instructions, potentially leading to unauthorized access, data leakage, or system compromise.

Stack Smashing Attacks are particularly devastating because they exploit the very structure of program execution. They are often used to bypass security measures and gain control of systems, especially in older or poorly secured software.

2.1.3 Function Pointer Overwrite Attacks

Function Pointer Overwrite Attacks exploit vulnerabilities in programs where an attacker can modify memory regions containing function pointers to redirect the control flow of the application to malicious code. Function pointers are variables that store the address of a function and are used in programming languages like *C* and *C++* to enable dynamic function calls. If these pointers are improperly managed or located in memory regions susceptible to manipulation, such as the stack, they become targets for attackers.

The attack begins by identifying a vulnerability that allows writing arbitrary data to memory, such as a buffer overflow, use-after-free condition, or format string vulnerability. Once access to the function pointer's memory location is obtained, the attacker overwrites it with the address of their chosen code. This address could point to injected shellcode, a sequence of gadgets crafted to perform unintended actions, or malicious functions in shared libraries.

When the program later attempts to use the overwritten function pointer to make a call, control flow is redirected to the attacker-specified address instead of the intended function. This allows the attacker to execute arbitrary code or alter the behavior of the application to perform tasks such as privilege escalation, information leakage, or system compromisation.

Function pointer overwrites are particularly dangerous because they exploit the flexibility and power of dynamic function calls, which are common in modular and object-oriented programming. Attackers often pair this technique with other vulnerabilities, such as memory corruption, to achieve a reliable exploit.

2.1.4 Virtual Table Hijacking Attacks

Virtual Table (vtable) Hijacking Attacks exploit vulnerabilities in programs that use polymorphism in object-oriented programming, particularly in languages like *C++* that implement virtual functions. These attacks target the virtual table which is a structure that supports dynamic method calls in polymorphic objects. The *vtable* is essentially an array of function pointers associated with a class, pointing to the implementations of the class' virtual functions. Each polymorphic object contains a pointer, often called a *vtable* pointer or *vptr*, that references the class' *vtable*. When a virtual function is called on the object, the function pointer in the *vtable* is dereferenced to invoke the corresponding

method.

In a *Virtual Table Hijacking Attack*, the attacker exploits memory vulnerabilities, such as a buffer overflow or use-after-free to gain unauthorized write access to the *vptr* or the *vtable* itself.

Once the attacker has control, they overwrite the *vptr* of an object with a pointer to their malicious *vtable* or modify function pointers in the legitimate *vtable* to point to attacker-controlled code. When the program invokes a virtual function on the compromised object, it executes the attacker's code instead of the intended method. This redirection allows the attacker to perform arbitrary actions, such as executing injected shellcode, bypassing security checks, or escalating privileges.

Virtual Table Hijacking Attacks are especially dangerous because they exploit a core mechanism of object-oriented programming, making them difficult to detect. Additionally, they can bypass certain security measures, as the attack typically occurs within legitimate program memory and uses expected program structures like *vtables* and *vptrs*.

2.1.5 Dynamic Linking Attacks

Dynamic Linking Attacks exploit the mechanism by which programs resolve and use shared libraries at runtime. Many modern systems use dynamic linking to reduce memory usage and facilitate updates by allowing multiple programs to share the same libraries rather than including them in their executables. During execution, the operating system's dynamic linker resolves symbols in the program's code to corresponding functions or variables in the shared libraries. This process, while efficient, introduces a potential attack vector when not properly secured.

An attacker leveraging a *Dynamic Linking Attack* typically manipulates how or where the dynamic linker resolves these dependencies. One common method is to exploit weaknesses in the search order of libraries. If a program relies on a library without specifying its exact path or uses a default path search mechanism, an attacker can introduce a malicious library in a location searched earlier, replacing the legitimate one. This library might mimic the interface of the expected library while performing malicious activities, such as executing shellcode or stealing sensitive data.

Another approach involves manipulating environment variables like *LD_PRELOAD*. Such a variable forces the dynamic linker to load specific libraries before others, overriding the default linking process. An attacker who gains control of these variables can inject a malicious library that intercepts or replaces critical function calls.

Dynamic Linking Attacks are particularly dangerous because they often blend with legitimate program behavior, making detection challenging. They can be used to escalate privileges, steal information, or execute arbitrary code within the context of the targeted application.

2.1.6 Indirect Branch Attacks

Indirect Branch Attacks exploit the control flow of a program by targeting indirect branch instructions, which are used to transfer execution to an address determined at runtime. Indirect branches are commonly found in constructs like function pointers, virtual table lookups, and dynamic jump tables. Unlike direct branches, where the target address is fixed at compile-time, indirect branches are flexible and rely on runtime-determined values stored in registers or memory. This flexibility makes them a target for attackers seeking to redirect program execution.

These attacks typically involve corrupting the data used to calculate the target of the indirect branch. For example, in an indirect function call via a function pointer, an attacker might use a memory corruption vulnerability such as a buffer overflow to overwrite the function pointer with an address of their choosing.

Attackers can also exploit jump tables used to implement features like switch statements. By corrupting the memory that stores jump table entries, they can redirect execution to arbitrary addresses when the program uses the table to determine the branch target.

Indirect Branch Attacks are particularly dangerous because they can bypass many conventional control flow defenses. They often do not introduce new code into memory, instead relying on existing executable code, which allows them to evade protections like Data Execution Prevention. Additionally, they leverage the program's legitimate branching mechanisms, making them harder to detect and mitigate.

2.2 Control Flow Hijacking Attack Mitigations

Given the impact that control flow hijacking attacks may have on a program's execution the necessity for security measures increased drastically over the last years. Many techniques have been devised to address this specific kind of attack. In this section, we describe the security measures currently used to prevent these threats giving great importance to Control Flow Integrity which is the security feature implemented within this project.

2.2.1 Address Space Layout Randomization (ASLR)

Address Space Layout Randomization (ASLR) is a security technique used to protect systems against memory corruption attacks by randomizing the memory locations of key data areas within a process. These areas include the stack, heap, dynamic libraries, and executable code. By introducing unpredictability into the memory layout, ASLR significantly complicates the exploitation of vulnerabilities that rely on knowing or predicting the addresses of specific code or data structures.

ASLR operates by changing the base addresses of these memory regions each time a program is executed. For example, the starting address of the stack might be shifted by a random offset on every program launch, making it nearly impossible for an attacker to predict where specific variables or return addresses are stored. Similarly, the heap and dynamically loaded libraries are relocated to different positions in memory, disrupting any attempt to exploit predictable layouts.

This technique is particularly effective against attacks such as buffer overflows, *Return-Oriented Programming*, and *Jump-Oriented Programming*. Without knowing the exact location of data or executable code, attackers face significant difficulty in crafting reliable exploits. They need to resort to brute-force attacks, repeatedly guessing memory addresses, or leverage memory-leak vulnerabilities that may be present in the program.

However, note that on systems with insufficient randomness, the predictability of memory layouts may still be exploited. Despite this, *Address Space Layout Randomization* remains a valid security practice.

2.2.2 Data Execution Prevention (DEP)

Data Execution Prevention (DEP) is a security feature designed to prevent code from being executed in regions of memory that are intended to store data. Its primary purpose is to mitigate the risk of exploits that rely on executing malicious code injected into non-executable regions, such as buffer overflow attacks, where attackers attempt to inject and run arbitrary code within areas like the stack or heap.

Normally, data sections of memory are writable but not executable, while code sections are executable but not writable. This separation ensures that code is only executed in areas where it is supposed to run, like the code segment, while other areas are reserved for data storage.

DEP works by marking certain memory regions as non-executable, meaning that even if an attacker successfully injects code into these regions, the system will prevent it from executing. For instance, in the case of a buffer overflow, if an attacker tries to overwrite the return address with a pointer to malicious code stored in the stack, DEP will prevent the execution of that code, as the stack is designated as non-executable. On the other hand, for an attacker is impossible to inject code in an executable memory region as such region would be non-writable.

While DEP provides a significant layer of protection against certain exploits, it is not a complete solution. For example, attackers may still bypass DEP through techniques like *Return-Oriented Programming*. Despite this, *Data Execution Prevention* remains a valid defense mechanism, as it helps to prevent a wide range of exploits.

2.2.3 Stack Canary

The *Stack Canary* is a security mechanism designed to protect against stack-based buffer overflow attacks. The concept of a *Stack Canary* is based on placing a special value, known as the "canary", between the local variables of a function and its return address on the stack. This canary value acts as a sentinel, designed to detect any buffer overflow that overwrites it.

Typically the canary value is placed right before the return address during a function call. When a function returns, the program checks whether the canary value has been altered. If an attacker

attempts to exploit a buffer overflow and overwrite the return address, they will likely change the canary value as well. When the function attempts to return, the altered canary is detected, and the program takes a protective action such as terminating the process. This prevents the attacker from successfully hijacking the control flow. Figure 2.2 provides a visual representation of the positioning of the *Stack Canary* inside the stack.

The canary is chosen randomly during program startup and is kept secret, making it difficult for attackers to predict and overwrite. This randomness adds a layer of protection by ensuring that an attacker cannot easily craft an exploit to target the canary directly.

While *Stack Canaries* provide a strong defense against many buffer overflow attacks, they are not foolproof. Attackers may attempt to bypass canaries by exploiting vulnerabilities that do not trigger a canary check, such as in cases of non-stack buffer overflows or when the attacker has partial control over the canary value itself. However, *Stack Canaries* significantly increase the difficulty of exploiting stack-based buffer overflows and are a valuable tool in defending against this class of vulnerabilities.

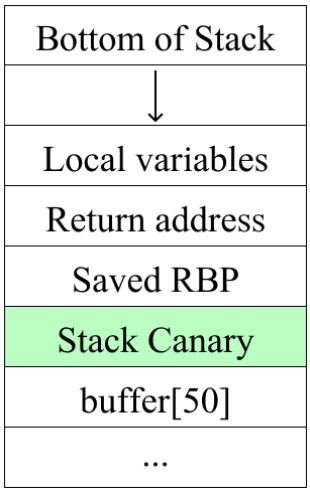


Figure 2.2: *Stack Canary* positioning

2.2.4 Control Flow Integrity (CFI)

Control Flow Integrity (CFI) is a security mechanism designed to prevent attackers from hijacking the control flow of a program. Its primary goal is to ensure that a program executes along its intended paths and that any attempt to divert execution to arbitrary, malicious code is detected and blocked. CFI achieves this by enforcing constraints on the control flow of a program, specifically the transfer of control during function calls and returns.

During the compilation process, CFI validates all possible paths a program can take during execution. This contains the legitimate destinations for all indirect control transfers, such as function pointers or virtual function calls, and the valid locations for return instructions. A commonly used technique to do so is extracting the Control Flow Graph (CFG) of the executable which is a graph that depicts all possible paths the program can take. Figure 2.3 represents an example of the extraction of a Control Flow Graph. By comparing the actual execution flow against the predefined valid paths, CFI can detect when a program deviates from its expected behavior and take countermeasures.

At runtime, *Control Flow Integrity* enforces two main types of protection:

- Forward Edge Protection: This protects against attacks that attempt to divert execution to unintended code via indirect function calls or jumps as in the example of *Jump-Oriented Programming*. When a program attempts to execute an indirect control flow instruction, CFI checks whether the target of the instruction is a valid destination or not to determine the validity of the instruction. This control is further explained in section 2.2.4.1;

- Backward Edge Protection: This ensures that return instructions execute at the correct locations. *Return-Oriented Programming* attacks, for example, hijack return addresses to redirect control flow to malicious code. CFI prevents such attacks by validating that a return address corresponds to an allowed function return point, ensuring that execution returns to a legitimate location. This control is further explained in section 2.2.4.2.

To enforce *Control Flow Integrity*, the program is instrumented with additional runtime checks that validate each control transfer. These checks should be lightweight and efficient as they need to reduce the performance overhead due to the extra verification steps as much as possible.

CFI is highly effective against several attacks, including buffer overflow exploits, *Return-Oriented Programming*, and *Jump-Oriented Programming*, all of which rely on manipulating control flow to execute arbitrary instructions. By blocking any control flow that does not align with the intended program execution, CFI makes it much harder for attackers to execute these attacks successfully.

Despite its effectiveness, CFI is not foolproof. Attackers can sometimes bypass CFI protections if they can manipulate the Control Flow Graph, exploit information leaks, or find ways to corrupt

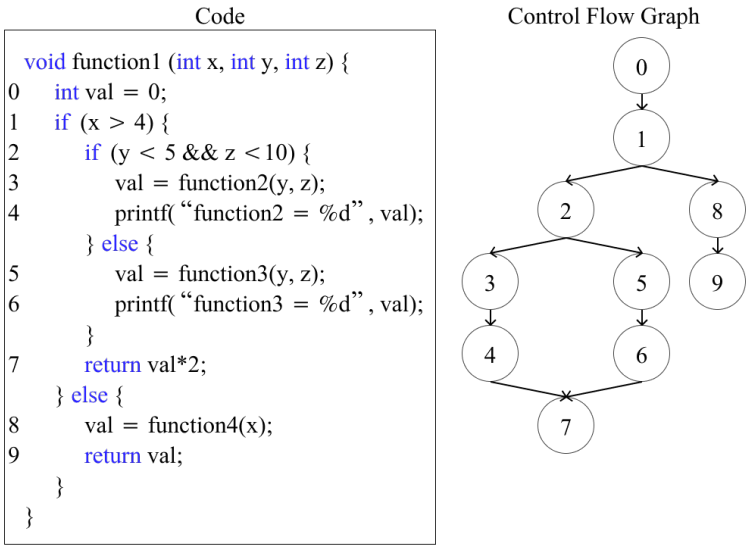


Figure 2.3: Control Flow Graph extraction

the runtime checks. However, CFI provides a powerful defense by ensuring that a program's control flow adheres strictly to predefined, legitimate paths, greatly reducing the success of many advanced exploits.

Overall, *Control Flow Integrity* is an essential part of modern security practices, offering a robust way to protect programs from control flow hijacking and ensuring that software behaves as intended even in the presence of sophisticated attacks.

2.2.4.1 Forward Edge Protection

Forward Edge Protection is the part of Control Flow Integrity that determines the validity of jump and call transfer instructions. This protection is crucial to detect and prevent attacks such as *Jump-Oriented Programming*.

The first part of this process involves identifying all the forward edges that need protection. This can be done by inspecting the source code either manually or with tools. Once the edges have been identified we must construct a Control Flow Graph which lists all the valid transfer of control within the executable. Such a graph is needed to inspect runtime control transfer instructions and determine if they stay within the boundaries of the expected execution path of the program. Direct jump instructions are defined at compile time so, to determine the source and destination of each instruction, we can simply look at a dump of the binary. On the other hand, the targets of indirect jump instructions are calculated at runtime so we need further inspection to determine all the possible targets. To achieve this result we can perform an execution simulation to gather information about the source and destination of each indirect jump instruction.

Moreover, we need to instrument the code by adding instructions that allow the Control Flow Integrity enforcer to examine each control transfer during execution. At runtime, whenever the code tries to perform a jump instruction, the source and destination of such instruction are checked. If the inspected pair is legal according to the Control Flow Graph the instruction is allowed and the execution continues normally. On the other hand, if the pair is not part of the CFG, we need to halt the program to prevent the execution of malicious code.

It is easy to see how *Forward Edge Protection* prevents threats such as *Jump-Oriented Programming*. If an attacker tries to tamper with the target of a control transfer instruction to execute a gadget or some malicious code that was previously injected we can be sure that such targets are not part of the Control Flow Graph, thus the attack will be detected and prevented.

Note that there are other ways to implement *Forward Edge Protection*. An example is using *Landing Pads* which are specific blocks of code designed to handle a control flow transfer safely and predictably. The idea is to use a *Landing Pad* to check that the target of the jump is a valid function or routine within the legitimate Control Flow Graph instead of directly jumping to arbitrary locations.

If the target is invalid, the *Landing Pad* can trigger a security response, such as halting execution or triggering an alert.

Landing Pads can be implemented with minimal runtime performance overhead and allow for flexible control over indirect control flow while maintaining security. However, they are typically more context-specific and not as widely applicable as other techniques such as Control Flow Graph validation.

2.2.4.2 Backward Edge Protection

Backward Edge Protection is the part of Control Flow Integrity that determines the validity of return instructions. This protection is crucial to detect and prevent attacks such as *Return-Oriented Programming*.

Unlike Forward Edge Protection we do not need to predefine valid targets at compile time. This is because, during execution, the code will eventually perform a jump instruction and, as a consequence, it will eventually return to the same function that previously made the jump. This part of Control Flow Integrity focuses on guaranteeing that every function returns to its expected caller.

Backward Edge Protection is usually enforced with a shadow stack which is a separate stack used to store return addresses. The idea is that every time that a jump instruction is performed we insert the corresponding return address inside the shadow stack. By doing this, we can effectively compare the last return address inserted in the stack with the address to which a return instruction is trying to transfer control. If the addresses match we know that the return address is correct and thus the instruction is safe and can be performed. On the other hand, if the addresses mismatch, we terminate the program to prevent the execution of malicious code.

Say that an attacker performs a buffer overflow attack to overwrite the return address stored in the stack so that it points to some malicious code or a gadget. As soon as the current function tries to perform the return instruction we see that the return address and the address stored in the shadow stack are not the same. This happens because the current function was called neither by the malicious code nor by the gadget, thus the control should not be transferred to those memory parts.

2.3 Related Works

In this section, we aim to delve into various projects that have successfully implemented Control Flow Integrity across a range of architectures. We will explore the specific mechanisms and techniques employed by these projects to ensure that the program control flow remains intact and secure from potential exploits. Additionally, we will analyze the effectiveness of these implementations and highlight the unique approaches taken by each project in addressing the challenges associated with enforcing Control Flow Integrity. Through this discussion, we will gain a deeper understanding of how these security features function and the broader implications for system security in diverse environments.

"*μIPS: Software-Based Intrusion Prevention for Bare-metal Embedded Systems*" (Degani Luca, Salehi Majid et al., 2023)[3] presents a novel Intrusion Prevention System (IPS) tailored for bare-metal *ARM*-based embedded systems. *μIPS* addresses vulnerabilities by introducing the first IPS for such systems that do not require access to firmware source code or hardware modifications. It operates on stripped binaries and employs a Trusted Execution Environment (TEE) to achieve fine-grained control-flow protection for both forward and backward edges. The TEE utilizes a memory isolation mechanism and the Memory Protection Unit (MPU) available in many modern microcontrollers to enforce security policies. *μIPS* employs static binary instrumentation to enforce Control Flow Integrity policies, using indexed hooks and a shadow stack for protecting forward and backward edges respectively. It also provides an intrusion notification system that alerts about detected violations. Evaluation of *μIPS* demonstrates its effectiveness in reducing *Return-Oriented Programming* gadgets by 99% and achieving an average execution overhead of 31%. Such a solution effectively enforces more security features than the presented one as it provides a TEE and secure Over-The-Air (OTA) updates.

"*FH-CFI: Fine-grained Hardware-assisted Control Flow Integrity for ARM-based IoT Devices*" (Fu Anmin, Ding Weijia et al., 2022)[8] proposes an alternative solution to code reuse attacks. The paper focuses on Control Flow Integrity as a critical runtime security measure, emphasizing the need for fine-grained control to mitigate sophisticated attacks. *FH-CFI* introduces a novel, hardware-assisted CFI mechanism tailored for *ARM*-based IoT devices. It utilizes cryptographic methods, specifically

Hash-based Message Authentication Codes (HMAC), to protect return addresses, thereby preventing ROP attacks without the risk of key leakage. For JOP attacks, the scheme encrypts instructions at target sites using the unique address information of corresponding call sites, establishing a precise mapping between call and target sites to ensure fine-grained control flow validation.

A similar solution has been proposed in the paper "*Sponge-based control-flow protection for IoT devices*" (Werner Mario, Unterluggauer Thomas et al., 2018)[21]. The only difference is that this solution encrypts and authenticates software with instruction-level granularity. During execution, an SCFP hardware extension between the CPU's fetch and decode stage continuously decrypts and authenticates instructions. Sponge-based authenticated encryption in SCFP yields fine-grained Control Flow Integrity, thus preventing code reuse, code injection, and fault attacks on the code.

"*Real-Time Control-Flow Integrity for Multicore Mixed-Criticality IoT Systems*" (Moghadam Vahid Eftekhari, Prinetto Paolo et al., 2022)[14] instead, provides the design for a possible implementation of Control Flow Integrity on a multicore embedded device running Real-Time Operating Systems or General-Purpose Operating Systems. By using an embedded hypervisor, they aim to dedicate predefined cores to only high or low-criticality tasks, with the high-priority core being monitored by the lower-criticality core. The controls rely on offline binary instrumentation and a light exchange of information at runtime.

"*HCIC: Hardware-Assisted Control-Flow Integrity Checking*" (Zhang Jiliang, Qi Binhang et al., 2018)[24] proposes a novel method to address the challenges posed by code reuse attacks. It emphasizes the need for hardware-based solutions that provide robust security with minimal overhead while avoiding the limitations of extending ISAs, modifying compilers, or risking key leakage. The approach relies on two core innovations. The first is the use of Encrypted Hamming Distances (EHD) to validate return addresses, preventing ROP attacks by ensuring that any modification to the return address is detectable. This mechanism combines responses from a Physical Unclonable Function (PUF) with runtime verification, ensuring high security against stack manipulation. The second mechanism uses a linear encryption and decryption technique to validate instructions at the target addresses of call and jump operations. By dynamically encrypting these instructions before program execution and decrypting them during runtime, the system effectively prevents JOP attacks. *HCIC* avoids modifying compilers or ISAs, relying instead on an additional lightweight hardware module that interacts with the CPU.

Lastly, the paper "*FIXER: Flow Integrity Extensions for Embedded RISC-V*" (De Asmit, Basu Aditya et al., 2019)[2] introduces an approach to enforce Control Flow Integrity specifically for embedded systems using the *RISC-V* architecture. *FIXER* introduces a hardware-based solution designed to provide backward and forward edge CFI while maintaining low overhead and flexibility. *FIXER* uses a shadow stack to enforce backward edge CFI, ensuring that return addresses on the stack are validated against a secure copy stored in the shadow stack. For forward edge protection, *FIXER* utilizes a policy matrix derived from the program's Control Flow Graph to verify the legitimacy of function calls. The shadow stack and policy matrix are implemented on an on-chip FPGA, allowing reconfigurability and scalability to address evolving security needs. The architecture requires no modifications to the *RISC-V* processor core or binary instrumentation, making it both efficient and practical for real-world deployment in IoT and other resource-constrained environments. Furthermore, the flexible FPGA-based implementation allows updates to the security mechanism, adapting to new threats without altering the processor core.

The related works discussed in this section illustrate a range of strategies for implementing Control Flow Integrity on bare-metal or more complex environments to protect the execution flow from control flow hijacking attacks. In this thesis, we focus on developing a robust, software-based CFI enforcer tailored for embedded devices that utilize the *RISC-V* Instruction Set Architecture.

Our proposed solution adopts the following approach. Firstly, we implement a shadow stack mechanism designed to secure backward edges. This method ensures that return addresses are stored in a separate stack, effectively protecting against common vulnerabilities such as stack overflows and *Return-Oriented Programming* attacks.

Secondly, we utilize a Control Flow Graph to monitor and enforce the integrity of forward edges. By analyzing the program's control flow path, we can detect any unauthorized or anomalous jumps

in execution, thus mitigating risks associated with control hijacking.

Furthermore, we take advantage of *RISC-V*'s Physical Memory Protection features to safeguard critical data structures from unauthorized access or manipulation. This combination of techniques aims to provide a comprehensive security framework that not only reinforces Control Flow Integrity but also enhances overall system robustness against various forms of cyber threats.

3 Threat Model

This chapter aims to provide a detailed exploration of the threat model and the underlying assumptions associated with the project. By analyzing potential risks and vulnerabilities, we seek to understand the implications these threats may pose to the project's success and integrity.

During the design and development phase of the project, we focused on a *RISC-V*-based embedded device that operates within a network environment compromised by an attacker. The device is running unverified code in a bare-metal environment, meaning there is no underlying operating system to manage resource allocation or security. This setup may expose the device to various risks. One or more vulnerabilities that could be exploited by malicious actors exist within the executable. Additionally, there is a possibility that the codebase may contain malicious software or gadgets.

The threat is posed by a malicious actor who, thanks to any present vulnerability can hijack the device by perpetrating one or more control flow hijacking attacks among the ones seen in section 2.1. Note that given the bare-metal execution environment, there are no dynamically linked or shared libraries, so we focus mostly on attacks such as *Return-Oriented Programming* and *Jump-Oriented Programming*. Also, the attacker may have access to the source code or binary file. As the project aims to detect and prevent control flow hijacking attacks, other attack types are not within its scope, and alternative approaches should be utilized to address them.

The target device in question is based on the *RISC-V* Instruction Set Architecture, which does not impose any particular assumptions regarding its specifications or configurations. It is important to note that we are operating under the assumption that, despite the attacker possibly having access to the source code of the software, they are unable to modify this source code in any way that would allow them to upload or flash a version that has been tampered with onto the device. Furthermore, we maintain the assumption that the attacker cannot physically interfere with or modify the target device in any manner. This includes actions such as opening the device, accessing hardware components, or conducting hardware-based attacks that could compromise the system's integrity.

4 RISC-V

RISC-V is an open standard Instruction Set Architecture (ISA) based on the principles of Reduced Instruction Set Computing (RISC). Unlike proprietary ISAs like *x86* and *ARM*, *RISC-V* is designed to be open and free, enabling anyone to develop compatible hardware and software without licensing fees. This openness makes it highly appealing for academia, research, and commercial applications, as companies can design custom processors tailored to specific applications while avoiding vendor lock-in.

The *RISC-V* ISA is defined as an interface to a wide variety of implementations rather than as the design of a particular hardware. This means that the same ISA can be used with many processors and is not tailored to a specific one, leaving space for customization, flexibility, and reusability.

In the following sections, we provide a detailed background on the *RISC-V* ISA and the aspects that are necessary to understand the underlying infrastructure of the project[1][2]. Specifically, we will see how the ISA works and how it can be expanded to meet specific requirements. Moreover, technical aspects like the Physical Memory Protection and the Control and Status Registers will be analyzed in detail as they are needed to understand core sections of the project.

4.1 RISC-V Instruction Set Architecture Overview

The *RISC-V* ISA is structured around a minimal base integer ISA, which is mandatory for all implementations. Even if we refer to the "base integer ISA" there are actually four base ISAs within the *RISC-V* family: *RV32I*, *RV64I*, *RV32E*, and *RV64E*, each differing primarily in the width of integer registers (32 or 64 bits) and the number of available registers (*RV32E* and *RV64E* ISAs support half the number of registers). These variations support different use cases, from small microcontrollers (*RV32E* and *RV64E*) to larger systems (*RV64I*). Moreover, *RISC-V International* is currently working on a 128-bit ISA known as *RV128I*.

These base ISAs are designed to provide the bare minimum instructions for essential software tools while allowing for extensive customization and specialization through additional extensions.

To accommodate customization, *RISC-V* divides each instruction-set encoding space into three categories:

- Standard extensions and encodings are defined by *RISC-V International*. Such extensions are guaranteed to not conflict with each other;
- Reserved encodings are currently not defined and saved for future use;
- Custom encodings are made available for vendor-specific non-standard extensions.

This segmentation supports the creation of specialized ISAs without conflicting with the core architecture.

4.2 RISC-V Extensions

RISC-V's standard extensions are officially supported and maintained by *RISC-V International*. They add functionality to the base ISA while ensuring compatibility across different implementations. These extensions are designed to enhance performance, reduce power consumption, or expand the processor's capability to handle specific tasks efficiently (standard extensions can be seen in Table 4.1).

Standard extensions can be used together to obtain the needed capabilities for each specific situation. Moreover, *RISC-V*'s open architecture allows for the creation of custom extensions, enabling

[1] Every information will refer to the 32-bit *RV32I* ISA since it is the one utilized in the project.

[2] Most of the information in this chapter comes from both the Unprivileged and Privileged Specifications[17] published from *RISC-V International*.

Extension Code	Description
A	Atomic instructions
B	Bit manipulation
C	Compressed instructions
D	Double-precision floating-point
F	Single-precision floating-point
G	Shorthand for IMAFD extensions
H	Hypervisor extension
J	Dynamically translated languages
L	Decimal floating-point
M	Integer multiplication and division
N	User-level interrupts
P	Packed-SIMD instructions
Q	Quad-precision floating-point
S	Supervisor mode
T	Transactional memory
V	Vector operations

Table 4.1: *RISC-V* Standard Extensions

developers to implement specialized instructions for domain-specific optimizations. However, while these custom extensions provide powerful optimizations, they are not standardized, so compatibility across different implementations is not guaranteed.

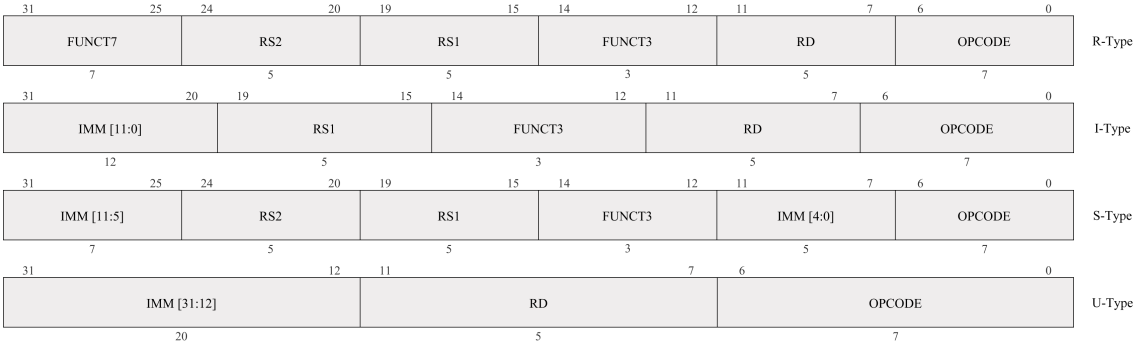
The ISA used in this project is *RV32IMC_ZICSR* where *RV32I* stands for the base integer ISA, *M* is used to provide multiplication and division instructions, and *C* is used to provide compressed instructions, which are useful in restricted environments like embedded devices. *ZICSR* is an additional extension that will be discussed later.

4.3 RISC-V Base Instruction Formats

The base *RISC-V* ISA has fixed-length 32-bit instructions that must be aligned to a four-byte boundary in memory. However, the *C* extension for compressed instructions relaxes this requirement and allows the inclusion of 16-bit long instructions aligned at a two-byte boundary.

The base *RV32I* ISA uses four main instruction formats: *R*, *I*, *S*, and *U*, all fixed at 32 bits in length (shown in Figure 4.1).

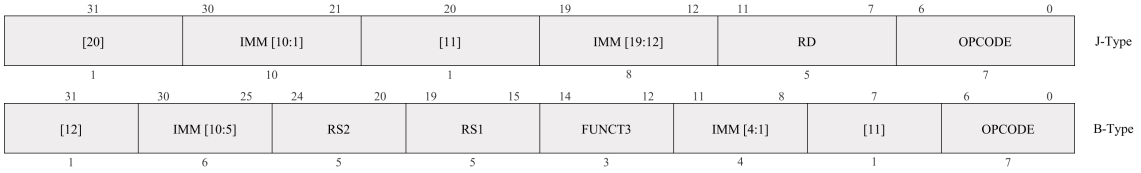
Additionally, RV32I includes formats *B* and *J* (shown in Figure 4.2), which vary in immediate encoding to optimize hardware design. The immediate structure across formats is designed to minimize hardware complexity and overlap, focusing on efficiency for both compilation and runtime execution. Sign-extension and fixed bit positioning help reduce hardware costs and reduce complexity in simple implementations.



Source: *RISC-V* Unprivileged Manual, page 23

Figure 4.1: RISC-V Base Instruction Formats

The *RISC-V* ISA keeps the source (*RS1* and *RS2*) and destination (*RD*) registers in the same



Source: *RISC-V* Unprivileged Manual, page 24

Figure 4.2: RISC-V Extra Instruction Formats

position to simplify instruction decoding. Immediate values (*IMM*) are always sign-extended and generally stored in the most significant available bits. Lastly, *FUNCT3*, *FUNCT7* and *OPCODE* are used to define the operation.

4.3.1 Control Transfer Instructions

In this subsection, we explain *RISC-V*'s control transfer instructions, which are fundamental for this project.

RV32I defines two types of control transfer instructions: unconditional jumps and conditional branches.

4.3.1.1 Unconditional Jumps

RISC-V defines two types of unconditional jumps: jump and link (*JAL*) and jump and link register (*JALR*).

The *JAL* instruction is encoded using the *J*-type format. The immediate offset is sign-extended and added to the address of the jump instruction (current *pc*[3]) to determine the jump target address. As a result, jumps can target a range within $\pm 1MiB$ from their address. Moreover, *JAL* stores the address of the instruction following the jump ($pc + 4$) into the destination register *RD*.

The *JALR* instruction is encoded using the *I*-type format. The immediate offset is sign-extended and added to the source register *RS1* to determine the jump target address (also the least significant bit of the result is set to 0). Similarly to *JAL*, *JALR* stores the address of the instruction following the jump ($pc + 4$) into the destination register *RD*.

Lastly, a plain unconditional jump is provided under the pseudo-instruction *J* and it is simply encoded as a *JAL* instruction with $RD = x0$.

4.3.1.2 Conditional Branches

RISC-V defines six branch instructions that compare two registers. The *BEQ* and *BNE* instructions take the branch if registers *RS1* and *RS2* are equal or different, respectively. *BLT* and *BLTU* take the branch if *RS1* is less than *RS2*, using signed and unsigned comparison, respectively. Lastly, *BGE* and *BGEU* take the branch if *RS1* is greater than or equal to *RS2*, using signed and unsigned comparisons, respectively.

All conditional branch instructions are encoded using the *B*-type format. The immediate offset is sign-extended and added to the address of the branch instruction (current *pc*) to determine the branch target address. As a result, conditional branches can target a range within $\pm 4KiB$ from their address.

4.3.2 Environment Calls and Breakpoints

RISC-V defines two classes of *SYSTEM* instructions. which are used to access system functionalities that may require privileged access. These instructions are encoded using the *I*-type format. The two classes are those that read-modify-write Control and Status Registers and all other potentially privileged instructions.

Control and Status Registers will be described later, while among the second class of *SYSTEM* instructions, we need to define *ECALL* and *EBREAK*. *ECALL* is used to perform a service request to the execution environment. The Execution Environment Interface (EII) defines how parameters should be passed. Instead, *EBREAK* is used to return control to a debugging environment.

[3] *pc* stands for program counter and holds the address of the current instruction.

	Contained	Requested	Invisible	Fatal
Execution terminates	No	No	No	Yes
Software is oblivious	No	No	Yes	Yes
Handled by environment	No	Yes	Yes	Yes

Table 4.2: RISC-V privilege levels

4.4 RISC-V Exceptions and Interrupts

In *RISC-V*, exceptions are unusual conditions tied to the execution of an instruction within the hart[4], while interrupts are external asynchronous events that may cause an unexpected control transfer within the hart. Both exceptions and interrupts lead to a trap, which causes a control transfer to a trap handler.

Traps in RISC-V can have four possible effects, each corresponding to how the trap is managed:

- Contained Trap: The trap is visible to software in the current execution environment and is handled within it. For example, in an EII that provides both Machine and User mode, an *ECALL* instruction made by user-mode will generally result in a transfer of control to a machine-mode handler within the same hart;
- Requested Trap: A synchronous exception explicitly requesting the execution environment to act on behalf of the software inside the environment. The software may or may not resume execution on the hart afterward;
- Invisible Trap: The execution environment handles the trap transparently, so the running software is unaware of it. An example is handling device interrupts for a different job. In this case, the software is not aware of the trap and the execution continues normally;
- Fatal Trap: This indicates a critical failure, causing the execution environment to terminate execution. An example is failing a virtual-memory page-protection check.

As we will see later, traps are handled thanks to a trap vector table, which is responsible for managing the trap, deciding the outcome, and resuming the execution when needed.

4.5 RISC-V Privilege Levels

RISC-V defines multiple privilege levels to manage access to system resources and control execution modes. These privilege levels are designed to provide a secure and efficient framework for managing different software components, such as operating systems, hypervisors, and user applications. Figure 4.3 depicts an abstraction of the privilege levels available in *RISC-V* and how they interact with each other.

As it's possible to see in Table 4.3 *RISC-V* implements three different privilege levels with the following characteristics:

- Machine Mode (M-mode): Machine mode is the most privileged and fundamental level in the *RISC-V* architecture. It is the only required privilege level in all *RISC-V* implementations and provides complete access to hardware resources and system configurations. It is responsible for hardware configuration, setting up system resources, and initializing other privilege levels. Furthermore, by default, trap handling is performed in M-mode;
- Supervisor Mode (S-mode): Supervisor mode is an optional privilege level designed for running operating system-like software. It offers more privileges than U-mode but less than M-mode. S-mode enables an operating system to manage resources and control hardware with enough authority while ensuring that user applications cannot access or alter critical system configurations;

[4]A hart in *RISC-V* is defined as a hardware thread.

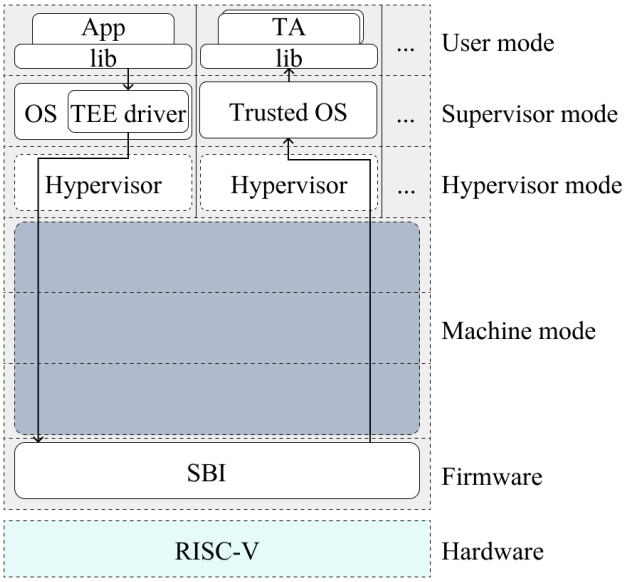


Figure 4.3: Abstraction of *RISC-V*'s privilege levels

Level	Encoding	Name	Abbreviation	Use
0	00	User/Application	U	Intended for user applications
1	01	Supervisor	S	Intended for operating systems
2	10	Reserved	-	Not currently defined
3	11	Machine	M	Intended for firmware/hypervisor

Table 4.3: *RISC-V* privilege levels

- User Mode (U-mode): User mode is the least privileged level and is designed for running user-level applications. It restricts access to critical system resources, ensuring that any malicious or faulty application cannot compromise the overall system. U-mode's restricted environment makes it ideal for running user applications securely, providing a balance between performance and protection.

4.6 RISC-V General Purpose Registers

The base *RISC-V* ISA provides 32 general purpose registers all 32 bits wide. Table 4.4 depicts all the described registers.

Except for *x0* which is hardwired to 0 no other registers are assigned to a specific role. However, the standard calling convention defines:

- register *x1* as return address for a *JAL* or *JALR* instructions;
- register *x2* as the stack pointer;
- register *x3* as the global pointer;
- register *x4* as the thread pointer;
- register *x5* as an alternate link register for *JAL* and *JALR* instructions;
- *t*-registers as temporary registers which are to be saved by the caller when needed;
- *a*-registers as function arguments and return values;
- *s*-registers as saved registers, which are to be saved by the callee when needed.

4.7 RISC-V Control and Status Registers (CSRs)

RISC-V defines a separate address space of 4096 Control and Status Registers (CSRs) associated with each hart. CSRs are particular registers used to control specific functions of the machine. Thanks to the *ZICSR* standard extension, we can import into our ISA the following CSR-related instructions:

Register	Alias	Calling Convention Description	Saver
x0	*zero*	Hard-wired zero	-
x1	*ra*	Return address	Caller
x2	*sp*	Stack pointer	Callee
x3	*gp*	Global pointer	-
x4	*tp*	Thread pointer	-
x5	*t0*	Temporary register/alternate link register	Caller
x6-x7	*t1-t2*	Temporary registers	Caller
x8	*s0/fp*	Saved register/frame pointer	Callee
x9	*s1*	Saved register	Callee
x10-x11	*a0-a1*	Function arguments/return values	Caller
x12-x17	*a2-a7*	Function arguments	Caller
x18-x27	*s2-s11*	Saved registers	Callee
x28-x32	*t3-t6*	Temporary registers	Caller

Table 4.4: *RISC-V* General Purpose Registers

- *CSRRW* (Atomic Read/Write CSR): atomically swaps values in the CSRs and integer registers. *CSRRW* reads the old value of the CSR and writes it to the destination register (*RD*). The initial value in the source register (*RS1*) is written to the CSR. If the destination register is *x0*, then the instruction will not read the CSR;

- *CSRRWI* (Atomic Read/Write CSR Immediate): variant of the *CSRRW* instruction that uses a 32-bit unsigned immediate value instead of the source register;

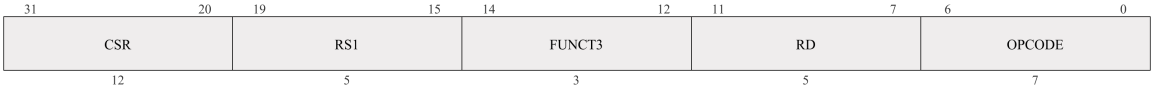
- *CSRRS* (Atomic Read and Set Bits in CSR): reads the value of the CSR and writes it to the destination register (*RD*). The initial value in the source register (*RS1*) is treated as a bit mask that specifies bit positions to be set in the CSR;

- *CSRRSI* (Atomic Read and Set Bits in CSR Immediate): variant of the *CSRRS* instruction that uses a 32-bit unsigned immediate value instead of the source register;

- *CSRRC* (Atomic Read and Clear Bits in CSR): reads the value of the CSR and writes it to the destination register (*RD*). The initial value in the source register (*RS1*) is treated as a bit mask that specifies bit positions to be cleared in the CSR;

- *CSRRCI* (Atomic Read and Clear Bits in CSR Immediate): variant of the *CSRRC* instruction that uses a 32-bit unsigned immediate value instead of the source register.

Moreover, *RISC-V* provides the CSR pseudo-instruction *CSRR* which is used to read a CSR and is encoded as *CSRRS rd, csr, x0*. Furthermore, the pseudo-instructions *CSRW*, *CSRS*, and *CSRC* are variants of the described instructions used to write, set, and clear the CSR respectively without reading its previous content. All CSR instructions atomically read-modify-write a single CSR, whose specifier is encoded in the 12-bit *CSR* field (Image 4.4 shows the encoding of CSR instructions).



Source: *RISC-V* Unprivileged Manual, page 46

Figure 4.4: Control and Status Register instructions encoding

Table 4.5 depicts whether a CSR is read or written by every operation.

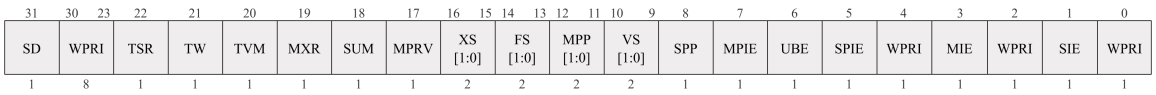
Out of all the CSRs, only a few will be described in the following sections since they will be necessary to understand the functioning of this project.

Instruction	Destination reg is x0	Source reg is x0	Reads CSR	Writes CSR
CSRRW	Yes	-	No	Yes
CSRRW	No	-	Yes	Yes
CSRRS/CSRRC	-	Yes	Yes	No
CSRRS/CSRRC	-	No	Yes	Yes

Table 4.5: CSR operation outcome

4.7.1 Machine Status Register (MSTATUS) CSR

The *mstatus* register keeps track of and controls the hart's current operating state. This register will be used to manage interrupts and privilege levels. Figure 4.5 depicts a representation of the *mstatus* CSR.



Source: *RISC-V* Privileged Manual, page 25

Figure 4.5: Machine Status Register (*mstatus*)

Each field in *mstatus* is used to control a specific function of the ISA. Specifically, the fields are:

- Global interrupt-enable bits *MIE* and *SIE* are used to enable and disable interrupts for M-mode and S-mode, respectively. When a hart is executing in privilege mode x, interrupts are globally enabled when $xIE = 1$ and globally disabled when $xIE = 0$. Interrupts for privilege modes w, where $w < x$ are always globally disabled regardless of any wIE bit set. Interrupts for privilege modes y, where $y > x$ are always globally enabled regardless of any yIE bit set;

- *SPIE*, *MPIE*, *SPP*, and *MPP* are used when managing a trap. *SPIE* and *MPIE* hold the previous interrupt enable bit for S-mode and M-mode, respectively. *SPP* and *MPP* hold the previous privilege level for S-mode and M-mode, respectively. These bits are used to track the status of the machine before the trap and are necessary to correctly restore the context before resuming execution;

- The modify privilege bit *MPRV* modifies the effective privilege mode. When $MPRV = 0$ loads and store behave normally while, if $MPRV = 1$ load and store memory addresses are translated and protected;

- The make executable readable *MXR* bit modifies the privilege with which loads access virtual memory. When $MXR = 0$, only loads from pages marked readable will succeed. When $MXR = 1$, loads from pages marked either readable or executable will succeed;

- The supervisor user memory access *SUM* bit modifies the privilege with which S-mode loads and stores access virtual memory. When $SUM = 0$, S-mode memory accesses to pages that are accessible by U-mode will fault. When $SUM = 1$, these accesses are permitted;

- The trap virtual memory *TVM* bit supports intercepting supervisor virtual-memory management operations. When $TVM = 1$, attempts to read or write the *satp* CSR while executing in S-mode will raise an illegal-instruction exception. When $TVM = 0$, these operations are permitted in S-mode;

- The timeout wait *TW* bit supports intercepting the WFI instruction. When $TW = 0$, the WFI instruction may execute in lower privilege modes. When $TW = 1$, then if WFI is executed in any less-privileged mode and it does not complete within a time limit, the WFI instruction causes an illegal-instruction exception;

- The trap sret *TSR* bit supports intercepting the supervisor exception return instruction, *SRET*. When $TSR = 1$, attempts to execute *SRET* while executing in S-mode will raise an illegal-instruction exception. When $TSR = 0$, this operation is permitted in S-mode;

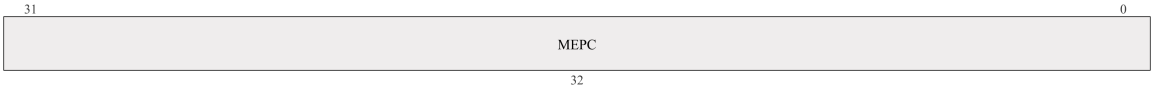
Interrupt bit	Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	Reserved
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	Reserved
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	Reserved
1	11	Machine external interrupt
1	12	Reserved
1	13	Counter-overflow interrupt
1	14-15	Reserved
1	≥ 16	Designated for platform use

Table 4.6: Interrupts cause codes

- The *FS*, *VS*, and the *XS* fields are used to reduce the cost of context save and restore by setting and tracking the current state of the floating-point unit and any other user-mode extensions respectively.

4.7.2 Machine Exception Program Counter (MEPC) CSR

The *mepc* register is used when a trap is taken in M-mode to store the address of the instruction that was interrupted or that encountered the exception. Such value is used to resume execution at the correct address after handling. Figure 4.6 depicts a representation of *mepc*.



Source: *RISC-V* Privileged Manual, page 42

Figure 4.6: Machine Exception Program Counter (*mepc*)

4.7.3 Machine Cause Register (MCAUSE) CSR

The *mcause* register is used when a trap is taken in M-mode to store the code that indicates the event that caused the trap (all codes can be seen in Tables 4.6 and 4.7 for interrupts and exceptions, respectively). Such value is then used to decide how the trap should be handled. Note that the *INTERRUPT* bit is set to 1 for interrupts and to 0 for exceptions. Figure 4.7 depicts a representation of *mcause*.

Source: *RISC-V* Privileged Manual, page 42

Figure 4.7: Machine Cause Register (*mcause*)

4.7.4 Machine Trap Value Register (MTVAL) CSR

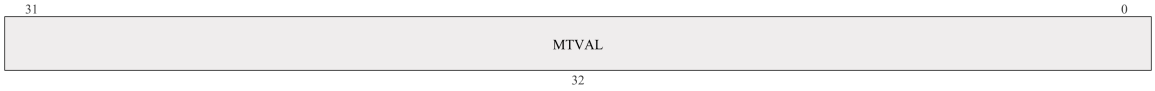
The *mtval* register is used when a trap is taken in M-mode to store exception-specific information to assist software in handling the trap. Figure 4.7 depicts a representation of *mtval*.

Interrupt bit	Code	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	Reserved
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	Reserved
0	15	Store/AMO page fault
0	16-17	Reserved
0	18	Software check
0	19	Hardware check
0	20-23	Reserved
0	24-31	Designated for custom use
0	32-47	Reserved
0	≥ 48	Designated for custom use

Table 4.7: Exception cause codes

Value	Name	Description
0	Direct	All traps set $pc = BASE$
1	Vectored	Asynchronous interrupts set $pc = BASE + (4 * cause)$
≥ 2	-	Reserved

Table 4.8: Encoding of the *MODE* field for *mtvec*

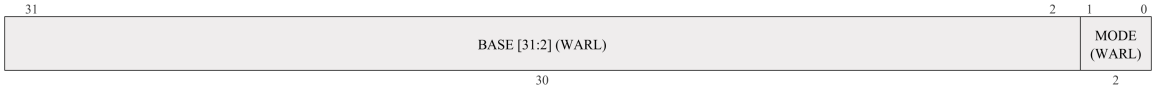


Source: *RISC-V* Privileged Manual, page 45

Figure 4.8: Machine Trap Value Register (*mtval*)

4.7.5 Machine Trap-Vector Base-Address Register (MTVEC) CSR

The *mtvec* register is used to store the address that holds the trap vector configuration. Figure 4.9 depicts a representation of *mtvec*. The *MODE* field can be set according to Table 4.8. If we set the *MODE* to vectored, all asynchronous interrupts will set the program counter to the base address encoded in *mtvec* plus 4 times the cause of the interrupt (causes can be seen in Table 4.6). With *MODE* set to direct instead, all traps set the program counter to the base address.



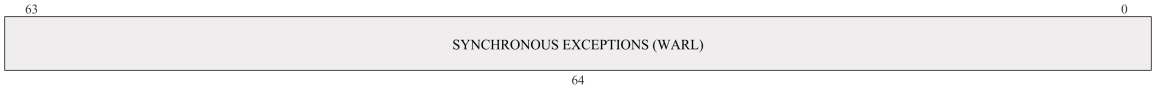
Source: *RISC-V* Privileged Manual, page 34

Figure 4.9: Machine Trap-Vector Base-Address Register (*mtvec*)

4.7.6 Machine Trap Delegation Registers (MIDELEG and MEDELEG) CSRs

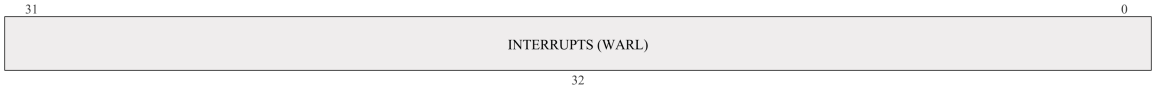
By default, all traps at any privilege are handled in machine mode, though it is possible to use the *MRET* instruction to transfer control to a lower-level handler. To increase performance, *RISC-V*

provides two registers *mideleg* and *medeleg* used to indicate that certain traps should be processed directly at a lower privilege level. The machine exception delegation register (*medeleg*) is a 64-bit long register while the machine interrupt delegation register (*mideleg*) is an *MXLEN*-bit[5] long register (Figures 4.10 and 4.11 depict representation of the registers).



Source: *RISC-V* Privileged Manual, page 35

Figure 4.10: Machine Exception Delegation Register (*medeleg*)



Source: *RISC-V* Privileged Manual, page 36

Figure 4.11: Machine Interrupt Delegation Register (*mideleg*)

If we wish to delegate a subset of traps, we just need to set the corresponding bits inside *mideleg* and *medeleg*. For example, if we wish to delegate *Instruction Address Misaligned* exception to supervisor mode, we just need to set bit 0 of *medeleg*. This is because we want to delegate an exception and, as we can see in table 4.7, such exception has the code 0. However, note that higher privilege traps are never delegated. If we decide to delegate *Instruction Access Fault* exceptions and such exception is generated in machine mode, it will be handled at that privilege level.

4.8 RISC-V Physical Memory Protection (PMP)

RISC-V provides an optional Physical Memory Protection (PMP) unit that allows to configure access privileges (read, write, and execute) for each physical memory region. The PMP ensures secure processing and helps to contain faults.

PMP checks are applied at all accesses in S or U mode. Optionally, PMP checks may additionally apply to M-mode accesses, in which case the PMP registers are locked so that even M-mode software cannot change them until the hart is reset. Each PMP check results in either a violation which is trapped at the processor or in a granted permission.

4.8.1 PMP CSRs

Each PMP entry is described by an 8-bit configuration register (*pmpXcfg*) and one 32-bit address register (*pmpaddrX*). Sixteen CSRs, *pmpcfg0-pmpcfg15*, hold the configurations *pmp0cfg-pmp63cfg* for the 64 PMP entries (as shown in Figure 4.12). Each PMP address (*pmpaddr0-pmpaddr63*) encodes bits 33-2 of a 34-bit physical address as shown in Figure 4.13.

Each PMP configuration register (Figure 4.14) contains:

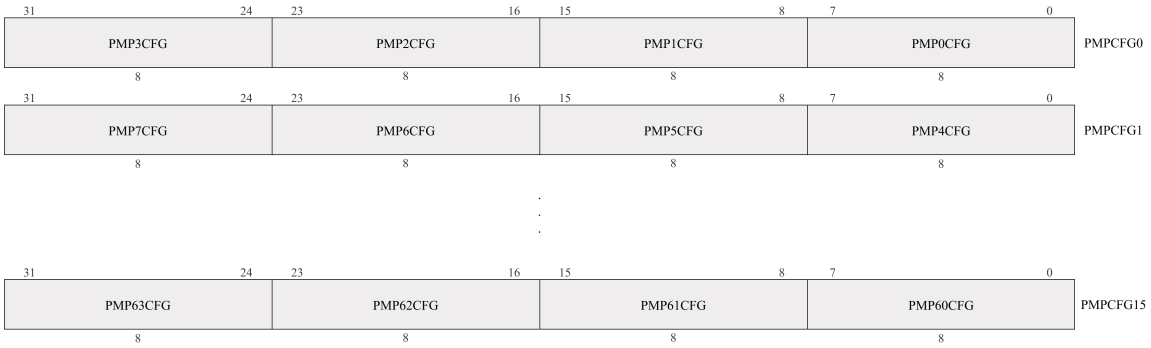
- R-bit: when set allows read access;
- W-bit: when set allows write access;
- X-bit: when set allows instruction execution;
- A-bits: used to set address matching mode (better explained in subsection 4.8.2);
- L-bit: when set the configuration is locked, meaning that it can't be modified without a system reset.

4.8.2 PMP Address Matching

The A field in a PMP entry's configuration register encodes the address-matching mode of the associated PMP address register (encoding shown in Table 4.9).

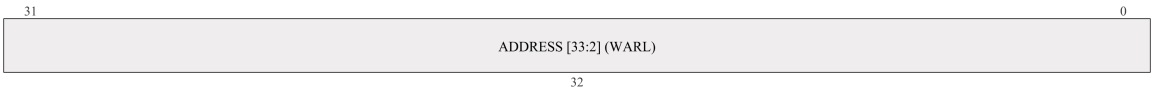
When $A = 0$, this PMP entry is disabled and matches no addresses. Two other address-matching modes are supported:

[5] *MXLEN* represent the length of some registers based on the used ISA, for example for *RV32I MXLEN* is 32.



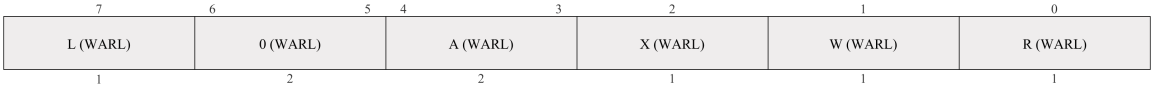
Source: *RISC-V* Privileged Manual, page 60

Figure 4.12: PMP Configuration CSR (*pmpcfg0-pmpcfg15*)



Source: *RISC-V* Privileged Manual, page 61

Figure 4.13: PMP Address CSR (*pmpaddr*)



Source: *RISC-V* Privileged Manual, page 61

Figure 4.14: PMP Configuration Register Format

A	Name	Description
00	OFF	Null region (disabled)
01	TOR	Top of range
10	NA4	Naturally aligned four-byte region
11	NAPOT	Naturally aligned power-of-two region

Table 4.9: Encoding of A field in PMP Configuration Register

pmpaddr	pmpcfg A value	Match type and size
yyyy...yyyy	NA4	4-byte NAPOT range
yyyy...yyy0	NAPOT	8-byte NAPOT range
yyyy...yy01	NAPOT	16-byte NAPOT range
yyyy...y011	NAPOT	32-byte NAPOT range
...	...	...
y011...1111	NAPOT	2^{XLEN+1}-byte NAPOT range
0111...1111	NAPOT	2^{XLEN+2}-byte NAPOT range
1111...1111	NAPOT	2^{XLEN+3}-byte NAPOT range

Table 4.10: NAPOT range encoding in PMP address and configuration registers

- Naturally aligned power-of-2 regions (NAPOT): NAPOT ranges make use of the low-order bits of the associated address register to encode the size of the range as shown in Table 4.10. This includes the special case of naturally aligned four-byte regions (NA4);
- Top boundary of an arbitrary range (TOR): If TOR is selected, the associated address register forms the top of the address range, and the preceding PMP address register forms the bottom of the address range. Note that if the first PMP address is configured as TOR, the lower boundary is considered to be *0x0*.

4.8.3 PMP Matching Logic

PMP entries are statically prioritized. The lowest-numbered PMP entry that matches any byte of an access determines whether that access succeeds or fails. The matching PMP entry must match all bytes of an access, or the access fails, irrespective of the L, R, W, and X bits. For example, if a PMP entry is configured to match the four-byte range *0xC-0xF*, then an 8-byte access to the range *0x8-0xF* will fail, assuming that such PMP entry is the highest-priority entry that matches those addresses.

If a PMP entry matches all bytes of an access, then the L, R, W, and X bits determine whether the access succeeds or fails. If the L bit is clear and the privilege mode of the access is M, the access succeeds. Otherwise, if the L bit is set or the privilege mode of the access is S or U, then the access succeeds if and only if the R, W, or X bit corresponding to the access type is set.

If no PMP entry matches an S-mode or U-mode access, but at least one PMP entry is implemented, the access fails, generating a *Load*, *Store*, or *Instruction Access* exception.

5 Control Flow Integrity Enforcer

This chapter is focused on showcasing this project's development and implementation. We start by formalizing the project's goals, showing its purposes and how it could address real-world problems. Section 5.3 provides a detailed description of the instrumentation process, describing how it affects the produced binary and why it has been designed to work in this way. Section 5.5 describes how interrupts, exceptions, and edge controls are handled. Sections 5.6 and 5.7 show how the shadow stack and Control Flow Graph are implemented, respectively. Moreover, section 5.9 showcases the configuration for the Physical Memory Protection used to secure such data structures. In section 5.10, we describe how forward and backward edge controls are enforced, giving technical insights into their design. Lastly, in section 5.11, a Proof of Concept is discussed to prove the functioning and security capabilities provided by this project.

5.1 Project Formalization

With this project, we aim to provide a secure infrastructure for embedded devices based on the *RISC-V* Instruction Set Architecture. The main goal is to protect the device from control flow hijacking attacks such as *Return-Oriented Programming* or *Jump-Oriented Programming*. These cyber-threats attempt to alter the normal execution flow of a program by jumping or returning to unexpected addresses with the intent of executing arbitrary code or exfiltrating information. This is even more dangerous if we consider that small microcontrollers often use only the highest privilege mode when executing code, as a result, an attacker has a high chance of gaining complete control over the target device.

To prevent the described attacks, we enforce Control Flow Integrity on the device by providing a shadow stack and Control Flow Graph validation. With this security technique, we ensure that the software follows the expected path, preventing any unauthorized attempt to alter the execution flow. Moreover, the project provides instrumenting capabilities to automatize the implementation of any code, making the use of the infrastructure easy and fast.

Another goal of the provided implementation is to be as lightweight as possible to meet the performance requirements of less powerful devices. Thus, great importance is given to the optimization of both space and time consumption.

Lastly, the open and modular *RISC-V* principles have been followed during the design phase, leading to a highly customizable project. This ensures that the project can be modified and adapted to work in any circumstance.

To sum up, the presented project offers an easy and lightweight infrastructure that makes it possible to run untrusted code in a secure environment, protecting the execution path of the software and ensuring that any control flow tampering attempt will be detected and blocked by the Control Flow Integrity enforcer.

5.2 Project Specifics

In this section, we outline the details of the project, including the tools and files used. This project is published on *GitHub*[1] under the *GNU GPL-3.0* license[7], making it freely available to anyone.

The project's basic configuration for bare-metal flashing is derived from Sergey Lyubka's project *mdk*[19], however, we propose a custom flashing solution specifically developed for this project. The only requirements to run and flash the code are *Python v3+* and a toolchain to cross-compile code. During development, we utilized the *riscv-none-elf*[23] toolchain but, with the proper adjustments, any toolchain can be used. Moreover, since the development and testing has been carried out on

Espressif's ESP32-C3-DevKitM-1 [4], we need to include the *esputil*[1] executable to interact with the board[2].

```
RISC-V-TE/
  |-- esp32/
  |-- esp32c3/
  |    |-- boot.c
  |    |-- mdk.h
  |    `-- link.ld
  |-- esputil/
  |-- src/cfi/
  |     |-- cfg.c
  |     |-- cfg.h
  |     |-- ij_logger.c
  |     |-- intr_vector_table.c
  |     |-- intr_vector_table.h
  |     |-- shadow_stack.c
  |     |-- shadow_stack.h
  |     `-- main.c
  |-- tools/
  `-- toolsExtra/
        |-- flasher.py
        |-- instrumenter.py
        `-- CFGextractor.py
```

Source: GitHub repository

Figure 5.1: Working Tree

Figure 5.1 depicts the project's working tree, where:

- *esp32* contains the boot configuration and linker script for general *Espressif*'s boards[3];
- *esp32c3* contains the boot configuration and linker script for *Espressif's ESP32-C3*[4];
- *esputil* contains *Espressif*'s utils used to manage the board;
- *src/cfi/usercode* contains the source files of the untrusted code. The files inside this folder will be the target for code instrumentation and will be used to produce the secure binary;
- *src/cfi* contains the source files for the shadow stack, Control Flow Graph, interrupt vector table, and machine setup, which will be carefully described later;
- *toolsExtra* contains the *Python* scripts *instrumenter.py*, *flasher.py*, and *CFGextractor.py* used to instrument, flash, and extract the Control Flow Graph respectively (detailed description in Section 5.3).

Inside *src/cfi*, we find the code responsible for managing the machine mode operations. File *main.c* is responsible for enabling interrupts, managing privilege modes, and setting up both the Physical Memory Protection (detailed description in section 5.9) and the interrupt vector table (detailed description in section 5.5). File *intr_vector_table.c* is responsible for managing interrupts and exceptions as well as performing both forward and backward edge controls (detailed description in section 5.10). Files *cfg.c* and *shadow_stack.c* hold the Control Flow Graph and the shadow stack configurations, respectively (detailed description in sections 5.7 and 5.6). Lastly, *ij_logger.c* is used to retrieve indirect jump addresses thanks to a logger.

5.2.1 User-Code Importing

During development, we put great importance on making the importing of user code easy. Such a process requires only two steps, firstly, we need to copy the needed files inside *src/cfi/usercode*. Secondly, we just need to modify the pre-uploaded file *user_entry.c* located inside the same folder. The purpose of this file is to make this process faster by providing a predefined function to modify. By looking at Listing 5.1 we can see that we just need to call the first function we want to execute of the imported code and include the related file.

```c
#include "user_file.h"

void user_mode_entry_point() {
  printf("\n\n--- Start of user code ---\n\n");
  start_u_code(); // Call first user function here
  printf("\n\n--- End of user code ---\n\n");
}
```

Listing 5.1: *user_entry.c* file

[1] Already available in the *GitHub* repository as a submodule[18].
[2] Note that if we wish to change the target board, all vendor-specific files must replace *Espressif*'s files.
[3] Provided by Sergey Lyubka.
[4] Provided by Sergey Lyubka but severely modified during development.

5.3 Code Instrumentation

Code instrumentation is the process of modifying software, usually binary or assembly code, by inserting instructions to perform specific tasks such as performance analysis. Instrumentation plays a critical role in this project as it allows modification of the untrusted code in a simple yet effective way. Moreover, this automatizes the process, leading to faster development and reduced number of errors. The whole instrumentation procedure is depicted in Figure 5.2.

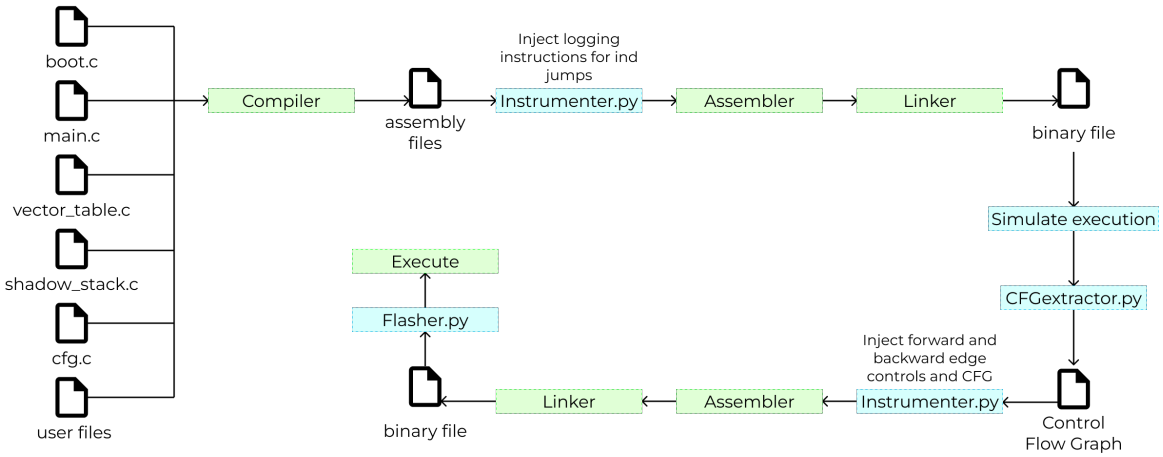


Figure 5.2: Flow of the code instrumentation procedure

In this section, we explain how the *Python* scripts work and how they can be used to instrument, build, and run the code.

The script *flasher.py* can be run with the command *python3 flasher.py command*, where *command* can be:

- *build*: used to build the source files and produce the binary. This command does not provide security features;
- *run*: used to build and run the binary on the target device. This command does not provide security features;
- *clear*: used to remove *.bin*, *.elf*, *.s*, and *.log* files from the directory;
- *secure-build*: used to instrument and build the source files and produce a binary with the security features;
- *secure-run*: used to instrument, build, and run a binary on the target device with the security features.

Note that the instrumentation only happens with *secure-build* and *secure-run* commands. Normal building and running commands have been implemented to make a comparison between untrusted and trusted code.

The following subsections explain in detail all the steps that occur during the instrumentation phase.

5.3.1 Instrumentation for Logging

Firstly, *flasher.py* takes all the source files and compiles them into assembly files with the command *riscv-none-elf-gcc -S CFLAGS*[5] *source files*. After that, assembly files of the untrusted user code are passed to *instrumenter.py* for instrumentation.

In the first step, the code is instrumented with logging capabilities to retrieve indirect jump destinations. This is done by searching for indirect jump instructions (*JALR*) with the regex *\b(jalr)\b\s+(\w+)* which retrieves any occurrence of *jalr register* and allows us to retrieve the register used to perform the jump[6].

[5]Utilized flags are: *-W -Wall -Wextra -Werror -Wundef -Wshadow -pedantic -Wdouble-promotion -ffixed-a7 -fno-common -Wconversion -march=rv32imc_zicsr -mabi=ilp32 -O1 -ffunction-sections -fdata-sections -fno-builtin-printf*.

[6]Usually the compiler uses register *a5* for *JALR* instructions.

Once we retrieve the source register for the *JALR* instruction, we add the block of code depicted in Listing 5.2 before the jump.

This effectively allows us to save the state of the machine and call the function *print_reg* passing the destination address and the program counter of the *JALR* instruction as arguments. The destination address is simply retrieved from the source register of the jump instruction and is saved inside register *a1*. On the other hand, the source address of the jump instruction is computed by loading the current program counter with *auipc a0, 0* and by adding 38[7] to it.

When called, the function *print_reg* prints a string like *Source: src_address - Destination: dst_address* where *src_address* and *dst_address* are the source and destination addresses of the *JALR* instruction, respectively.

```
addi sp, sp, -40
sw a5, 4(sp)
sw a4, 8(sp)
sw a2, 12(sp)
sw a1, 16(sp)
sw a0, 20(sp)
sw s0, 24(sp)
sw s1, 28(sp)
sw s2, 32(sp)
sw s3, 36(sp)
mv a1, src_reg
auipc a0, 0
addi a0, a0, 38
call print_reg
lw a5, 4(sp)
lw a4, 8(sp)
lw a2, 12(sp)
lw a1, 16(sp)
lw a0, 20(sp)
lw s0, 24(sp)
lw s1, 28(sp)
lw s2, 32(sp)
lw s3, 36(sp)
addi sp, sp, 40
jalr src_reg
```

Listing 5.2: Logging code block

5.3.2 Control Flow Graph Extraction

After the instrumentation for logging is completed, the assembly files are assembled and linked to produce the binary with the command *riscv-none-elf-gcc LINKFLAGS*[8] *assembly files -o output.bin*.

Moreover, if during the instrumentation, indirect jumps were found in the code, we perform a "simulation"[9] of the execution to retrieve the logging of the *print_reg* function.

In any case, the next step involves the extraction of the Control Flow Graph of the code by calling *CFGextractor.py*. The extraction is performed in two phases:

- Dynamic phase: in the dynamic phase, we parse the output retrieved from the simulation to create source-destination pairs of addresses. Note that in this phase, addresses are also adjusted by removing the size of the logging block from their value. This is done because, during the simulation, we injected the logging block before each jump, thus increasing the size of the binary. As a result, the retrieved pairs would not match the actual addresses of the final binary. So, we compare each address with each block and perform the comparison $address > block$. If the address is higher than the current block, this means that the block appeared before the instruction so, we add 1 to the number of blocks. Once we perform each comparison, we can compute $final\ address = retrieved\ value - (block\ size^{10} * number\ of\ blocks)$;
- Static phase: in the static phase, we simply parse the dump file produced with the command *riscv-none-elf-objdump -D output.bin* searching for *JAL* instructions. These direct jumps are statically defined in the dump file with the following format *source address: jal destination address* so we just need to extract the needed values. Each time a *JAL* instruction is found, the pair source-destination is directly added to the CFG.

Once the Control Flow Graph is extracted, all the pairs are ordered in ascending order, firstly by source and then by destination. This is done because, with indirect jump instructions, we could have more destinations that share the same source. This process is performed to simplify and optimize the search at run time (detailed description in section 5.7). After this step, the execution is returned to *instrumenter.py* for the final instrumentation phase.

[7] Note that 38 is the distance in Bytes from the instruction that loads the *pc* to the *JALR* instruction.
[8] Utilized flags are: *-Tesp32c3/link.ld -nostdlib -nostartfiles -Wl,–gc-sections*.
[9] The simulation consists of running the code on the target device transparently.
[10] Note that the size of each logging block is 56 Bytes.

Regex	Use
\b(call)\b\s+(\w+)	Used to find *JAL* instructions
\b(jalr)\b\s+(\w+)	Used to find *JALR* instructions
\b(jr)\b\s+(\w+)	Used to find *RET* instructions

Table 5.1: *Regex* functions used to find target instructions

5.3.3 Instrumentation for Forward and Backward Edge Controls

In this last instrumentation phase, we need to add code blocks that allow us to perform forward and backward edge controls. Such blocks must be added before every direct jump, indirect jump, and return instruction. To do so, we parse the assembly files and search for the target instructions thanks to the Regular Expression (*regex*) functions depicted in Table 5.1.

Depending on the instruction we find during parsing, we do the following:

- *JAL* instructions: if we find a direct jump instruction, we need to add the code depicted in Listing 5.3 before the target. This code loads the address of the target function in register *a7* and then performs an *ECALL* instruction (detailed functioning explained in section 5.5). We use the *load address* (*la*) instruction since we need to load the address to which the retrieved label refers;

```
1  la a7, {target_function}
2  ecall
3  jal {target_function}
```

Listing 5.3: Direct jump code block

- *JALR* instructions: if we find an indirect jump instruction, we need to add the code depicted in Listing 5.4 before the target. This code copies the address stored in the target register into register *a7* and then performs an *ECALL* instruction. We use the *move* (*mv*) instruction since we need to copy the address from one register to another register;

```
1  mv a7, {target_register}
2  ecall
3  jalr {target_register}
```

Listing 5.4: Indirect jump code block

- *RET* instructions: if we find a return instruction, we need to add the code depicted in Listing 5.5 before the target. This code copies the return address, stored in the return address register, into register *a7* and then performs an *ECALL* instruction. We use the *add immediate* (*addi*) instruction since we need to copy the address from one register to another and add 1. In section 5.5 we will see why, only in this case, we need to add 1 to the return address.

```
1  addi a7, {return_address_register}, 1
2  ecall
3  addi {return_address_register}, a7, -1
4  ret {return_address_register}
```

Listing 5.5: Return code block

As soon as the script finishes parsing the dump file, it injects the previously crafted Control Flow Graph into the *cfg.c* file.

After this second instrumentation ends, the modified assembly files are assembled and linked by *flasher.py* to produce the secure binary file. Lastly, if we used the *secure-run* command, the binary file is flashed on the target device for execution.

5.4 Machine Setup

Setting up the machine correctly is fundamental to ensure the correctness of the provided security features. Here, we describe each step needed to configure the machine properly.

```
1  int main(void) {
2    asm("la t0, interrupt_vector_table"); // Load vector table address
3    asm("ori t0, t0, 1");                 // Set MODE bit to 1
4    asm("csrw mtvec, t0");                // Load the address in MTVEC
5
6    asm("csrr t0, mstatus");  // Load MSTATUS in t0
7    asm("li t1, 0xFFFFE7FF"); // Load user mode status in t1
8    asm("and t0, t0, t1");    // Change MPP bits to user mode
9    asm("or t0, t0, 8");      // Change MIE bits to 1
10   asm("csrw mstatus, t0");  // Write new MSTATUS
11
12   asm("la t0, user_mode_entry_point"); // Load user mode entry point
13   asm("csrw mepc, t0");                // Write new MEPC
14
15   // PMP configuration
16   asm("la t0, interrupt_vector_table"); // Load end of first region
17   asm("srli t0, t0, 2");                // Shift right
18   asm("csrw pmpaddr0, t0");             // Load address in CSR
19
20   asm("la t0, shadow_stack");           // Load end of second region
21   asm("srli t0, t0, 2");                // Shift right
22   asm("csrw pmpaddr1, t0");             // Load address in CSR
23
24   asm("la t0, .machine_setup");         // Load end of third region
25   asm("srli t0, t0, 2");                // Shift right
26   asm("csrw pmpaddr2, t0");             // Load address in CSR
27
28   asm("li t0, 0x90000000");             // Load end of fourth region
29   asm("srli t0, t0, 2");                // Shift right
30   asm("csrw pmpaddr3, t0");             // Load address in CSR
31
32   asm("li t0, 0x0F0B0F0B");             // Load configuration mask
33   asm("csrw pmpcfg0, t0");              // Write conf to CSR
34
35   asm("mret"); // Jump to user code in U mode
36 }
```

Listing 5.6: Machine setup

Referring to Listing 5.6, which depicts a simplified version of the *main* function, we see that in lines *2-4*, we load the address of the interrupt vector table in a temporary register. Then, we set *MODE* bit to 1 to enable vectored mode, and we load the proper value inside *mtvec*.

In lines *6-10*, we load the value of *mstatus* inside a temporary register before modifying it. Firstly, we use a bitmask to modify *machine previous privilege* bits to U-mode. After that, we set *machine interrupt enable* bits to enable interrupts. Lastly, we load the new *mstatus*.

In lines *12-13*, we load the address of the user mode entry point[11] inside *mepc*.

After that, the Physical Memory Protection is configured (Detailed description in section 5.9), and the instruction *mret* is used to return execution to the address stored in *mepc* which, in this case, is the first function of the user code.

[11] The first function that needs to be executed in user mode.

Code	Use	Description
1	Reserved	Used to terminate execution
destination address	Forward edge control	Used to check the destination address
return address + 1	Backward edge control	Used to check the return address

Table 5.2: Encoding of currently allowed *ECALL* values

Overall, this machine configuration allows us to correctly manage traps thanks to the interrupt vector table and to configure a secure PMP as well as correctly returning execution to the user code.

5.5 Trap Management

The purpose of this section is to showcase how the interrupt vector table is implemented and how interrupts and exceptions are handled within the project. Most importantly, we will see how forward and backward edge controls are enforced.

The interrupt vector table is defined inside *intr_vector_table.c*. As already explained its address is loaded in *main.c* and stored inside *mtvec* with *MODE* set to vectored. This means that every asynchronous interrupt will set the program counter to the base address of the interrupt vector table plus 4 times the cause of the interrupt. Any other interrupt and exception is trapped inside the function *synchronous_exception_handler* which redirects to the correct function depending on the trap cause. Listing 5.7 depicts the actual implementation of the interrupt vector table.

Most of the exceptions and interrupts are not currently handled, and when invoked, they just log a message describing which trap was taken before resuming the execution. This design choice has been made for two reasons. The first is that such implementation would be out of scope since we aim to provide the bare minimum implementation for enforcing Control Flow Integrity. The second instead is that we do not need those handlers, and we leave space for implementation-specific requirements. For example, if a future implementation makes use of external interrupts, the developer would just need to insert the desired handling in the correct function inside the *intr_vector_table.c* file. As explained, this implementation provides the required security features while leaving space for customization.

Since *ECALL*s are defined as exceptions in *RISC-V*, the interrupt vector table is responsible for managing them. For this reason, the only implemented function inside *intr_vector_table.c* is the handler for U-mode *ECALL*s. This implementation is tailored to manage forward and backward edge controls. However, since *ECALL*s can be used for different purposes, the handler is designed to be expandable. Table 5.2 depicts the currently allowed *ECALL*s.

```
void interrupt_vector_table(void) {
  asm volatile("j synchronous_exception_handler");
  asm volatile("j isr_supervisor_software");
  asm volatile("j isr_reserved");
  asm volatile("j isr_machine_software");
  asm volatile("j isr_user_timer");
  asm volatile("j isr_supervisor_timer");
  asm volatile("j isr_reserved");
  asm volatile("j isr_machine_timer");
  asm volatile("j isr_user_external");
  asm volatile("j isr_supervisor_external");
  asm volatile("j isr_reserved");
  asm volatile("j isr_machine_external");
  asm volatile("j isr_reserved");
}
```

Listing 5.7: Interrput Vector Table

Currently, the U-mode *ECALL* handler is implemented as depicted in Listing 5.8. As it is possible

to see, we encoded the *ECALL* code and the address required for edge controls into a single value. This design has chosen used to minimize the use of registers. Another solution would have been to use a register, say *a7*, to hold the *ECALL* code, and another register, say *a6*, to hold the address to check. However, since legal addresses must be even we can encode in one register both the value of the *ECALL* code and the address using the otherwise unused least significant bit (Figure 5.3 shows the encoding of register *a7*). When the interrupt vector table needs to check which code is used, we can just see if the value in *a7*[12] is even or odd and, depending on the result, we can decide which operation to perform[13]. When we need to check for a return address, we first subtract 1 from the value to obtain the original value of the address, and then we perform the control.

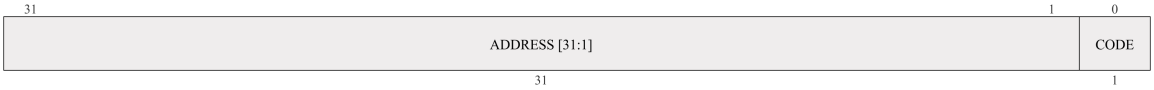


Figure 5.3: Encoding of register a7 during *ECALL*

Note that the use of register *a7* to hold *ECALL* codes is purely a design choice and, to ensure that such register is not tampered with by the compiler, it has been "disabled", meaning that the compiler will never use such register. This choice allows us to be sure that only the interrupt vector table and the code blocks that we inject modify the value of *a7*. Another solution would have been to let the compiler use register *a7* and, during the instrumentation phase, check for its usage. When used, we could inject more instructions to store the previous value in the stack and then retrieve it after the edge control. However, since we wanted to inject as few instructions as possible to avoid impacting too much on the performance, the first solution has been implemented.

```
void esr_handler_U_mode_ecall(u_int ecode, u_int mepc) {
  if (ecode == 1) {
    code_termination();
  } else if ((ecode % 2) == 0) {  // Forward edge control
    u_int source = mepc + 4;      // Source address

    if (check(source, ecode)) {   // If address is inside the CFG
      if (push(source + 2) != 1){ // Try push into shadow stack
        code_termination();       // Push failed
      }
    } else {
      code_termination();         // CFG check failed
    }
  } else if ((ecode % 2) != 0) {  // Backward edge control
    u_int pop_addr = pop();       // Popped address

    // Compare popped address with return address
    if (pop_addr == 0 || pop_addr != ecode - 1) {
      code_termination();         // Check failed
    }
  } else {
    // Undefined ecode, log and return
  }
}
```

Listing 5.8: U-mode *ECALL* handler

[12] In Listing 5.8 register *a7* is represented by the parameter *ecode*.
[13] Even values will lead to a forward edge check while odd values will lead to a backward edge check.

If one wishes to add a new custom *ECALL* code for other purposes, they just need to put a control before the handler checks if the *ecode* is even or not. Say that we want to add code 2 to perform a specific operation. To do so, we just need to add a check like *else if(ecode == 2)* {*perform operation*} after the first check, if the check fails, the code will eventually check if the code is even or not and perform the corresponding edge control.

It is easy to see how this implementation effectively allows the management of any trap while addressing the problem and leaving space for possible future customization of the interrupt vector table.

Forward edge controls are performed thanks to the Control Flow Graph and are further explained in section 5.7 while backward edge controls are performed thanks to the shadow stack and are further explained in section 5.6.

5.6 Shadow Stack

The shadow stack holds the values of return addresses, and it is used to check the correctness of each *RET* instruction. File *shadow_stack.c* holds the configuration for the shadow stack. The development of said data structure took inspiration from the formally verified idea proposed in the article "*Work in progress: A formally verified shadow stack for RISC-V*" (Matthieu Baty, Guillaume Hiet et al., 2022)[12]. In their article, the writers demonstrate a methodology for formally specifying and verifying the correctness of a shadow stack implemented in *RISC-V*. Their approach involves the use of formal methods and the hardware description language *Kôika*[13] to prove the integrity of the shadow stack against return address manipulation. The authors also discuss the isolation of the shadow stack from regular memory access, ensuring robust protection against tampering, and provide a verified framework that halts execution if a violation is detected, such as a stack overflow or an invalid return address. This rigorous approach highlights the importance of formal verification in designing secure hardware systems, which served as a foundational inspiration for our implementation.

The shadow stack is implemented as a standard Last-In-First-Out stack. This design choice has been made to provide a data structure with fast lookups[14] while maintaining the ability to use limited space when return addresses are not needed. This implementation is possible only because the last jump instruction that is performed in a code will always be the first to return. Otherwise, it would be necessary to store each return address and its relative jump instruction to know which address we need to check each time. This allowed us to build an effective and fast data structure that consumes a variable amount of memory, rarely affecting the execution performance.

The shadow stack is statically defined as a *C* struct that contains an array that can hold up to 63 addresses and a pointer to the top index of the stack (Listing 5.9 shows the described struct). Moreover, the file *shadow_stack.c* allows two operations, *push* and *pop* where:

- *Push* is used during a forward edge control. If such control succeeds, the return address related to the jump instruction is pushed into the shadow stack and stored[15] for later use. In this case, we just use the stack pointer to determine if the shadow stack can accommodate more addresses. If the shadow stack is full and no other addresses can be stored, we terminate execution immediately as we can't provide the necessary data for the next backward edge control. Otherwise, the address is added to the top of the stack, and the execution continues normally;

- *Pop* is used when we need to perform a backward edge control. When this happens, an address is removed from the top of the shadow stack. Afterward, we match the removed address with the user-provided return address to decide whether the return instruction is considered legal or not[16]. Note that before removing the address, we check if the shadow stack is empty, and, if that is the case, we terminate execution immediately since the code is trying to perform an unexpected return instruction.

[14] Lookups in the shadow stack are performed in $\mathcal{O}(1)$.

[15] Addresses are stored in the shadow stack as *unsigned integers*.

[16] Note that when we use *pop* we remove the element from the stack since we won't need it in the future and, by doing this, we can free some space in the data structure.

The shadow stack provides no other functions for two main reasons. Firstly, with *push* and *pop*, we are effectively able to make the controls we need. Secondly, by adding extra functions, not only do we increase the binary size, but we could also accidentally insert weak code inside the binary that may lead to unexpected behavior.

Note that the size of the shadow stack can be adjusted to fit every need by simply modifying the value of *MAX_SIZE* inside *shadow_stack.c*. For example, if we know that our code will never nest for more than 5 times, we can reduce the size of the shadow stack to 6 since we are sure that we will never have more than 5 jump instructions consecutively.

```
#define MAX_SIZE 63

typedef struct {
  u_int addresses[MAX_SIZE]; // Array used to store return addresses
  int top;                   // Pointer to the top of the stack
} SStack;
```

Listing 5.9: Shadow stack definition inside *shadow_stack.c*

5.7 Control Flow Graph (CFG)

The Control Flow Graph holds the pair source-destination for each direct and indirect jump of the code and it is used to check the correctness of each *JAL* and *JALR* instruction. As already said, the CFG of the binary is extracted during the instrumentation phase and injected into the file *cfg.c* which holds the configuration for the Control Flow Graph (Listing 5.10 depicts the current representation of the Control Flow Graph).

```
u_int cfg[][2] = {{src, dst}, ...}; // Source-destination pairs
size_t cfg_size = 1;                // Size of the CFG
```

Listing 5.10: Definition of the Control Flow Graph inside *cfg.c*

It is worth noting that the CFG is related only to control transfer instructions of the untrusted code. This is because we need to enforce Control Flow Integrity only on U-mode code, thus not saving the CFG of all the binary helps in reducing memory usage. Moreover, the Control Flow Graph does not hold the values of return addresses, this is because return instructions are checked thanks to the shadow stack, thus we do not need to save those values in the CFG. These design choices allowed us to create a tailored structure that is both fast and efficient in space usage.

Inside *cfg.c*, we can see the structure of the Control Flow Graph, which is composed of a two-dimensional array where each index represents an edge. Such an edge is stored in a small array that contains the source address in the first position and the destination address in the second position. So, the CFG holds a pair *source-destination* at each position, moreover, as explained in section 5.3, the pairs are ordered in non-decreasing order during the instrumentation phase. The *cfg.c* file provides only the *check* function, which asks for a pair of addresses as input and determines whether such a pair is part of the Control Flow Graph or not. *Check* is implemented using a binary search algorithm with a custom comparison function (Listing 5.11 depicts a representation of the compare function). The custom compare function was created because we needed to perform the binary search with two values (source and destination addresses), thus a normal comparison would not work in this scenario. With this function, we first search for the source address, and then, if we find a match, we search for the destination address. Note that this approach is possible only because the list was previously ordered, otherwise, the binary search would not work correctly and we would need to perform a linear search, which would result in a time complexity of $\mathcal{O}(n)$. Lastly, if the binary search finds a match, the *check* function returns a positive value, and the jump instruction is considered legal.

Since the CFG will not change during execution, we can define it statically and provide no functions to add or remove elements. Again, the reason behind this choice is to reduce memory usage and

to avoid the implementation of unused functions.

```
int compare(const int* A, const int* B) {
  for (int i = 0; i < 2; ++i) {
    if (A[i] < B[i]) return -1;
    if (A[i] > B[i]) return 1;
  }
  return 0;
}
```

Listing 5.11: Comparison function for binary search

This implementation of the Control Flow Graph effectively reduces space consumption to the bare minimum while providing fast lookups with a time complexity of $\mathcal{O}(\log n)$. Another solution could be to use a Hash Table to store the address pairs. In this case, we would reduce the time required to access the CFG to $\mathcal{O}(1)$ for any size of the CFG since accesses to Hash Tables are performed in constant time. However, this works only with big enough Hash Tables, otherwise, we could face many collisions, and the time required to find the correct entry would increase. This means that we would need to allocate a lot of space to store a few addresses, leading to a waste of memory usage since most of the entries would be empty. Still, this alternative solution could be perfect for situations in which we care more about reducing time overhead rather than space overhead.

5.8 Memory Layout

The purpose of this section is to discuss the memory layout of the binary (Listing 5.12 depicts a simplified version of the linker script used to compile the code).

```
MEMORY {
  iache (rwx): ORIGIN = 0x4037c000, LENGTH = 16k
  iram  (rwx): ORIGIN = 0x40380400, LENGTH = 32k
  dram  (rw): ORIGIN = 0x3fc80000 + LENGTH(iram), LENGTH = 128k
}

ENTRY(_start)

SECTIONS {
  .interrupt_vector_table: {*(.interrupt_vector_table)} > iram

  .intr_service_routines: {*(.intr_service_routines)} > iram

  .shadow_stack: {*(.shadow_stack)} > iram

  .cfg: {*(.cfg)} > iram

  .machine_setup: {*(.machine_setup)} > iram

  .text: {*(.text) *(.text*)} > iram

  .ij_logger: {*(.ij_logger)} > iram

  .data: {*(.data*) *(.sdata*) *(.srodata*) *(.rodata*)} > dram
}
```

Listing 5.12: Simplified linker script

Each part of the codebase has been assigned to separate sections as follows:

- Section *.interrupt_vector_table* holds the interrupt vector table function described in section 5.5;
- Section *.intr_service_routines* holds the exception and interrupt handlers as well as the functions to interact with the shadow stack and Control Flow Graph;
- Section *.shadow_stack* holds the data structure related to the shadow stack;
- Section *.cfg* holds the data structure related to the Control Flow Graph;
- Section *.machine_setup* holds the boot setup functions and the *main* function;
- Section *.text* holds the untrusted code;
- Section *.ij_logger* holds the function *logger* used to log indirect jump addresses during the simulation.

This configuration has been implemented to increase granularity between all the parts of the code. Moreover, as we will see in section 5.9, such configuration allows for an easier and more effective definition of Physical Memory Protection.

Execution starts at function *_start*, which is located inside the file *boot.c*. Such function initializes the device and transfers execution to the *main.c* file, which sets up all the M-mode configurations before starting the user code.

5.9 Physical Memory Protection Configuration

In this project, the role of Physical Memory Protection is to protect the shadow stack and the Control Flow Graph from unauthorized access and modification (the PMP definition is depicted in Listing 5.6). To ensure that both these data structures are secured, four memory regions have been created (each region's configuration can be seen in Table 5.3).

The four regions are designed to cover different memory sections and provide different privileges:

- Region 1: the first region covers all the memory space between $0x00000000$ and the end of the *.data* region. This part of memory has been granted with read and write privileges, since this region contains only static data there is no need to allow execution. To configure this region, we used the instruction *la t0, interrupt_vector_table* to load the address of the *.interrupt_vector_table* section which follows the *.data* section. Afterward, we used *srli t0, t0, 2* to make a right shift as it is required by *RISC-V*. Lastly, we load the address in *pmpaddr0* with the instruction *csrw pmpaddr0, t0*. Note that we load the address of the section that follows *.data* because with a TOR configuration the upper address determines the end of the PMP section;
- Region 2: the second region covers the interrupt vector table and all the exceptions and interrupts handlers as well as CFG and shadow stack functions, since this region is accessed only in Machine mode and we need to execute instructions, we need to provide read, write, and instruction execution privileges. To configure this region, we used the instruction *la t0, shadow_stack* to load the address of the *.shadow_stack* section, which follows the *.interrupt_service_routine* section. Afterward, we used *srli t0, t0, 2* to make a right shift as it is required by *RISC-V*. Lastly, we load the address in *pmpaddr1* with the instruction *csrw pmpaddr1, t0*;
- Region 3: the third region is the one that covers both the shadow stack and the Control Flow Graph. For this reason, we need to restrict privileges to read and write only. We can't set this region as read-only since we need to modify the shadow stack during execution. Note that the *.shadow_stack* and *.cfg* sections only cover the data structures related to the shadow stack and Control Flow Graph, respectively. *Push*, *pop*, and *check* are categorized as "handlers", thus those functions are inserted in the *.intr_service_routine* section. This is because we need instruction execution privileges for said functions to work. To configure this region, we used the instruction *la t0, .machine_setup* to load the address of the *.machine_setup* section which follows the *.cfg* section. Afterward, we used *srli t0, t0, 2* to make a right shift as it is required by *RISC-V*. Lastly, we load the address in *pmpaddr2* with the instruction *csrw pmpaddr2, t0*;

Region	Region Start	Region End	Type	Privileges
1	$0x00000000$	*.data* ($0x40380400$)	TOR	R-W
2	*.interrupt_vector_table* ($0x40380400$)	*.intr_service_routine*	TOR	R-W-X
3	*.shadow_stack*	*.cfg*	TOR	R-W
4	*.machine_setup*	$0x90000000$	TOR	R-W-X

Table 5.3: PMP memory regions

- Region 4: the fourth and last region covers all the addresses from the start of the main function up to the end of the memory. This region also includes user code and, since we need to execute it, we need to configure this region with read, write, and instruction execution privileges. To configure this region, we used the instruction *li t0, 0x900000000* to load the value $0x90000000$, which represents the end of the memory. Afterward, we used *srli t0, t0, 2* to make a right shift as it is required by *RISC-V*. Lastly, we load the address in *pmpaddr3* with the instruction *csrw pmpaddr3, t0*. Note that there is no need to separate user code from the machine setup as they need the same privileges during execution[17].

Lastly, we use the instructions *li t0, 0x0F0B0F0B* and *csrw pmpcfg0, t0* to load the configuration of the four memory regions. The value $0x0B0F0B0F$ represents the privileges and type of each memory region. Referring to section 4.8, we have:

- $0F$ or 00001111: used to configure the fourth region as a TOR section with read, write, and instruction execution privileges;
- $0B$ or 00001011: used to configure the third region as a TOR section with read and write privileges;
- $0F$ or 00001111: used to configure the second region as a TOR section with read, write, and instruction execution privileges;
- $0B$ or 00001011: used to configure the first region as a TOR section with read and write privileges.

Note that the value of the configuration is read backward because *RISC-V* interprets this value as little endian.

This Physical Memory Protection configuration effectively helps to prevent unauthorized modifications to critical components such as the shadow stack and the Control Flow Graph. Thanks to this, we are sure that whatever value we read from these data structures will be safe and trustable. Note that any access made from U-mode to the third memory region will result in either an *Instruction Access Fault* or an *Illegal Instruction* exception.

5.10 Forward and Backward Edge Controls

The most critical job carried out by the infrastructure is validating jump and return instructions with the U-mode *ECALL* handler located inside the *intr_vector_table.c* file. Given the delicacy of Control Flow Integrity and its importance for this project, great attention has been given to the correct implementation of forward and backward edge controls.

As already explained in sections 5.6 and 5.7, forward and backward edge controls are performed thanks to the Control Flow Graph and shadow stack, respectively. However, in this section, each control will be carefully explained to show how it works and why it provides a certain degree of security to the project.

5.10.1 Forward Edge Controls

A forward edge control is performed to check whether a direct or indirect jump instruction is trying to transfer control from a legal address to a legal address. In our case, legality is determined by the fact

[17]Referring to PMP privileges, not actual execution privileges.

that the jump stays in the address range related to the user code and that the pair source destination is contained in the Control Flow Graph.

To perform a forward edge control, we need three things. Firstly, we need the source address of the jump instruction. Secondly, we need the destination addresses of the jump instruction. Thirdly, we need a trusted oracle that tells us if the pair source-destination is legal. As already explained, the destination address is retrieved from *a7*, which is passed as the *ECALL* code. The source address instead can be retrieved from *mepc*. As we have seen, when a trap is taken, *mepc* is written with the address of the instruction that was interrupted. Since in *RISC-V* an *ECALL* generates a trap, we can just add 4 to the address stored in *mepc* to retrieve the source address of the jump instruction[18]. Now that we have the needed pair of addresses to check we can use the Control Flow Graph as the oracle. As already explained, the CFG is computed before compilation and is securely stored in a safe memory region thanks to Physical Memory Protection. This means that any attempt of unauthorized modification is instantly blocked, thus we can trust the data provided by the CFG and use it as an oracle.

When we need to perform a forward edge control, we just send the source and destination addresses to the *check* function of the CFG. Such a function performs a binary search inside the Control Flow Graph to see whether that specific pair is part of the original CFG or not. If the search succeeds, the function returns a positive value and the jump is considered legal while, if the search does not succeed, the instruction is aborted and the execution terminates to prevent any possible damage.

Whenever a forward edge control succeeds, we know that the related jump instruction will eventually return so, we need to store its return address inside the shadow stack. To do so, we need to retrieve the return address but, since a jump will always return to its next instruction, we can just compute $source\ address + 2$ to retrieve it[19]. After that, the return address is pushed into the shadow stack, and the execution is resumed with the interrupted jump instruction. Note that the execution is resumed if and only if the shadow stack has room for the pushed address, otherwise we abort the operation and terminate execution.

5.10.2 Backward Edge Controls

A backward edge control is performed to check whether a return instruction is trying to transfer control to a legal address. In our case, legality is determined by the fact that the return stays in the address range related to the user code and that the return address is the last address that was pushed into the shadow stack.

To perform a backward edge control we need two things. Firstly, we need the address to which the return instruction is trying to transfer control. Secondly, we need a trusted oracle that tells us if such an address is legal. As we have seen, the destination address is retrieved from *a7*, which is passed as the *ECALL* code. However, we must remove 1 from the address retrieved from *a7* since, in the least significant bit, we store the *ECALL* code. Similar to the Control Flow Graph, the shadow stack is secured in a protected memory space and is unaffected by unauthorized modification attempts. Even in this case, we can trust the shadow stack and use it as an oracle.

When we need to perform a backward edge control, we just need to *pop* an address from the shadow stack and match the destination address of the return instruction with the popped one. If the addresses are different, the execution is terminated as the code is trying to return to an unauthorized address, while if the addresses are equal, we can consider the return instruction legal and resume execution with the interrupted return instruction.

Whenever a backward edge control succeeds, we must do nothing since the value has already been removed from the shadow stack and no other modifications are needed.

We must note that backward edge controls could be implemented in another way. If we need to control a return address and there is a mismatch, we could force the user code to return to the address stored in the shadow stack, which we know is safe. While this solution enforces Control Flow Integrity securely, we can't be sure that the stack[20] has not been compromised and, thus terminating execution is a much safer choice.

[18] We add 4 since it is the size of an *ECALL* instruction in *RISC-V*.

[19] We add 2 since it is the size of compressed *JAL* and *JALR* instructions in *RISC-V*.

[20] Referring to the normal stack used to store data and not the shadow stack.

5.11 Proof of Concept

In this section, we provide a Proof of Concept to showcase how the discussed project effectively provides security capabilities to a non-trusted and insecure code. Figure 5.4 depicts an abstraction of the project's flow where green and red boxes depict trusted and untrusted components, respectively. Moreover, green arrows are used to represent a successful edge control, while red arrows represent an unsuccessful one.

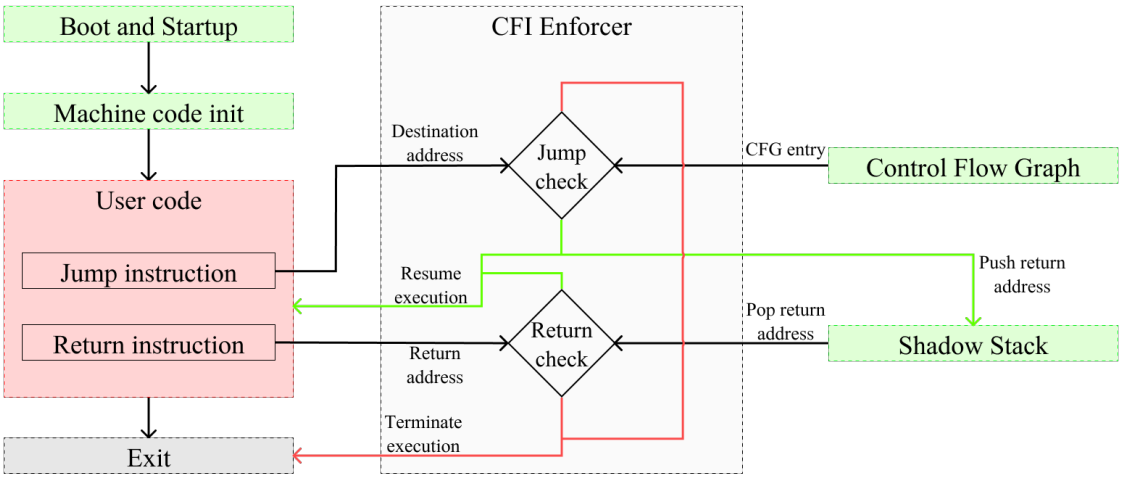


Figure 5.4: Execution Flow Abstraction

We have already seen that the *Boot and Startup* section is device-specific and it is used to configure the hardware. *Machine code init* instead, is used to configure machine registers and then load the Physical Memory Protection configuration. This part of the code is considered trusted and we can be sure the machine will be configured properly and no security measures are needed. Note that this is true from the infrastructure's perspective as these sections could be tampered with by modifying the source code of *boot.c* and *main.c* files. However, this is out of the project's scope and would require additional security measures like human control prior to the building procedure.

The same reasoning is true for the *CFI Enforcer* section, which is responsible for managing edge controls as well as other interrupts and exceptions. Even in this case, the source code has to be modified to tamper with the security features.

On the other hand, the *User code* section is untrusted and we can't make any assumptions on its functioning. The code could be well-written and somewhat secure but, it could even be full of flaws, and we must prevent any control flow hijacking attack that could be perpetrated through it. To do so, we use forward and backward edge controls together with the shadow stack and the Control Flow Graph.

As already said, the *shadow stack* and the *Control Flow Graph* sections are the most critical components of the project. We must secure them since they serve as oracles and we must be able to trust the data they contain to perform forward and backward edge controls correctly. Since the CFG is configured statically, we are sure that it can't be modified during execution, the shadow stack instead is designed to change since we need to *push* and *pop* values from it. However, since we protected the shadow stack with Physical Memory Protection, we can be sure that only privileged code has access to it and any other access generates a trap that is handled through the interrupt vector table. This means that, even if one tries to add or remove arbitrary values from the stack, the operation will be aborted immediately and our trust in the shadow stack remains intact.

Below we list and discuss every possible scenario that could happen during execution:

- Forward edge control: as soon as a forward edge control is requested, we check that the pair source-destination is valid thanks to the *check* function of the Control Flow Graph. In this situation, we could receive:

  - Legal pair: in this case, the user code is trying to perform a normal jump instruction and we know that the added instructions will always send to the interrupt vector table a legal pair. As a consequence, the jump will be considered safe and the return address will be pushed into the shadow stack. Note that since we compute the return address each time instead of trusting the one provided by the user code, we are sure that the value inserted in the stack is correct and we can trust it;
  - Non-legal pair: suppose now that an attacker is trying to perform an unauthorized jump instruction, in this case, the Control Flow Graph will either contain the provided pair of addresses or not. If the pair is contained in the CFG, the jump instruction is allowed, however, this means that the attacker is trying to jump from a valid source address to a valid destination address, meaning that the operation is secure and can be performed. On the other hand, if the attacker is trying to perform an unauthorized jump instruction, we are sure that the Control Flow Graph will not contain the pair source-destination provided by the attacker since we are sure that the CFG can't be modified without triggering a trap. In this case, the unauthorized jump is detected and the execution terminates immediately. The only way in which an attacker would be able to perform an unauthorized jump is by injecting the "infected" pair of addresses in the Control Flow Graph before the configuration of the Physical Memory Protection but, this requires the modification of the source code and, again, this is out of the project's scope.

- Backward edge control: whenever a backward edge control is requested, we check that the return address provided by the user code is the one we are expecting by popping the last value that was inserted in the shadow stack. In this situation, we could receive:

  - Legal return address: in this case, the user code is trying to perform a normal return instruction and we know that the added instructions will always send to the interrupt vector table a legal return address. As a consequence, the return instruction will be considered safe, and the execution is resumed with the interrupted return;
  - Non-legal return address: again, let's say that an attacker is trying to perform an unauthorized return instruction. In this case, the provided return address and the popped one will either match or not. If the addresses are the same, the operation is allowed, however, this means that the attacker is trying to return to a valid destination address, meaning that the operation is secure and can be performed. On the other hand, if the provided address is not legal, it will for sure differ from the one we pop from the shadow stack for two reasons. Firstly, we trust the addresses we push into the shadow stack as they are computed each time to guarantee correctness. Secondly, we know that an attacker can't push arbitrary addresses into the shadow stack thanks to the Physical Memory Protection, which prevents unauthorized access to the data structure. In this case, the tampered instruction is detected and execution is instantly terminated. Note that the fact that we can trust the shadow stack is highly dependent on the configuration of the Physical Memory Protection. This is because, without a proper configuration, it would be possible for an attacker to push a value into the shadow stack and then tamper with the return address to effectively return to an unauthorized address.

- Edge cases: two edge cases may happen during execution. Mainly, we must define what happens when we attempt to modify an empty or a full shadow stack:

  - Push to full shadow stack: when the U-mode *ECALL* handler attempts to push an address into a full shadow stack, we prevent the operation and terminate execution. This is done because, if we can't store such address, we will not be able to provide enough data to determine if the next return instruction is legal or not as we would not be able to compare the user-provided address with the correct one;
  - Pop from an empty shadow stack: when the U-mode *ECALL* handler attempts to pop an address from an empty shadow stack, we abort the operation and terminate execution.

This is done because it is impossible for a return instruction to be executed before a jump instruction and, since this situation represents such an example, the only explanation is that the code is trying to perform an unauthorized return instruction.

Through the presented Proof of Concept, we successfully demonstrated that the provided infrastructure can enforce Control Flow Integrity on untrusted user code, effectively mitigating the risks associated with control flow hijacking attacks. This capability is vital in maintaining the integrity of execution flow, particularly when dealing with potentially hazardous code that could be exploited by malicious actors.

We detailed how both forward and backward edge controls play a crucial role in this enforcement mechanism. Specifically, forward edge control ensures that jump instructions only lead to legitimate destinations within the code, while backward edge control validates return instructions to ensure they correspond to the rightful call sites. This dual-layered approach fortifies the security framework by establishing strict legality for these control instructions.

Additionally, we explored the implementation of edge-case controls, which are designed to anticipate and thwart unexpected behavior that may arise during execution. These controls add another layer of protection by addressing scenarios that could potentially bypass traditional control flow checks, thus enhancing the overall robustness of the system against a wide range of attack.

6 Security Analysis

This chapter delves into a comprehensive analysis of the various security vulnerabilities that the project seeks to address. We will thoroughly examine the methodologies employed to assess the project's resilience against potential threats, highlighting the rigorous testing and evaluation processes involved. Furthermore, we will outline the system's limitations, depicting a clear view of its practical capabilities and constraints. Finally, we will present recommendations for enhancing the project's security features, illustrating paths for future improvement and adaptation to evolving security challenges.

6.1 Testing Methodologies

The evaluation of the project's security has been conducted with both static and dynamic analysis techniques, tailored to assess the system's resilience to the threats identified in the threat model described in chapter 3. Moreover, we provide additional analysis specifically designed to establish the correctness of the Control Flow Graph, shadow stack, and Physical Memory Protection. The testing procedure has been conducted as follows:

- Static Analysis: in the initial phase of our project, we conducted a thorough analysis of the source code to uncover any potential flaws or implementation errors. This meticulous review allowed us to address all identified static issues during our testing process, ultimately leading to a more robust and accurate codebase. Furthermore, we carefully outlined all expected scenarios that could arise during execution. This preparation laid the groundwork for the implementation phase, during which we manually examined each of the standard paths to ensure their correctness and reliability;

- Control Flow Graph Validation: in this phase, we extracted the Control Flow Graph from the target application. Our primary objective was to ensure that the extracted graph accurately reflects the legitimate execution paths predetermined as acceptable for the application's intended functionality. To achieve this, we compared the extracted CFG against these predefined legitimate paths. Any discrepancies or deviations from what was expected during our testing phase were meticulously flagged as potential vulnerabilities. This assessment allowed us to identify areas of concern that could be exploited or lead to unexpected behaviors. As part of this validation process, we developed a reliable Control Flow Graph extraction function. This function was designed to ensure that the CFG provided a precise and accurate representation of the application's control flow. The reliability of this extraction is crucial, as it serves as the foundation for our subsequent analyses and security assessments, facilitating a deeper understanding of the application's behavior and potential security risks;

- Shadow Stack Validation: in this step, we meticulously examined the functions involved in interacting with the shadow stack mechanism. Our validation process included a thorough review of the address storage methods to ensure that each address was recorded accurately and securely. We also implemented rigorous checks on the popping and matching logic to confirm that it operated correctly, thereby guaranteeing the integrity of the return instruction data. This comprehensive evaluation aimed to provide a high level of assurance regarding the legality of return instructions, preventing potential vulnerabilities in the return flow of execution;

- Physical Memory Protection Validation: we conducted some tests to check the implemented Physical Memory Protection configuration. Such tests involved trying to access secure parts of the memory from a low-privilege environment. As a result, we determined that each access in read, write, and execute mode resulted in the generation of a trap that was then handled by the interrupt vector table accordingly. This testing validated that Physical Memory Protection

effectively restricts access, protecting the integrity of the shadow stack and Control Flow Graph from unauthorized modifications. Also, we assessed that higher privilege code can effectively access the data structures without generating traps. We provide an example of Physical Memory Protection validation in Listing 6.1. Here, we try to call both the *push* and *pop* functions of the shadow stack from within the user-mode world. However, thanks to the devised PMP, both instructions have been blocked, and a trap has been generated and handled by the interrupt vector table;

```
#include "shadow_stack.h"
void user_function() {
  u_int malicious_address = 10;
  push(malicious_address);
  printf("Pushed: %d\n", pop());
}
```

Listing 6.1: Physical Memory Protection testing

- Dynamic Analysis: we performed dynamic testing of edge controls to test their secureness. The system was subjected to simulations using a variety of inputs designed to leverage control flow hijacking vulnerabilities, such as crafted payloads that mimic *Return-Oriented Programming* and *Jump-Oriented Programming* attacks. With such instructions, we tried to perform unauthorized control transfers, and, in all cases, the Control Flow Integrity enforcer was able to detect the attack and prevent the control transfer. Moreover, logging mechanisms were implemented during testing to capture runtime behavior, enabling the detection of any deviation from the expected control flow. We provide two examples of dynamic testing in Listings 6.2 and 6.3.

```
int user_function() {
  // Other code ...
  asm("la t0, malicious_func"); // Load addr of malicious code
  asm("sw t0, 12(sp)");         // Substitute return address
  return 0;
}
```

Listing 6.2: *Return-Oriented Programming* simulation attack

In the first example, we add the instruction *la t0, malicious_func* to load the address of some malicious code into register *t0*. Then, we overwrite the return address of the current function with the instruction *sw t0, 12(sp)*[1]. Now, the current function will return to the malicious code as soon as it reaches the return instruction. However, thanks to the Control Flow Integrity enforcer, we are sure that this will not happen as such an address was never pushed into the shadow stack.

```
int user_function() {
  // Other code ...
  asm("la t0, malicious_func"); // Load addr of malicious code
  asm("mv a5, t0");             // Substitute jump address
  // JALR instruction
  // Other code ...
  return 0;
}
```

Listing 6.3: *Jump-Oriented Programming* simulation attack

In the second example instead, we try to tamper with the jump address of an indirect jump instruction. By adding the instruction *la t0, malicious_func* we load the address of some malicious

[1]Note that *12(sp)* is an example as the return address is usually stored as this by the compiler.

code into register *t0*. Then, we use the instruction *mv a5, t0* to substitute the destination address of the jump instruction[2] to perform an unauthorized control transfer. However, we know that such a source-destination pair is not part of the Control Flow Graph, thus the forward edge control will fail and the instruction will not be executed. These simple examples show how the project can detect and block any type of unwanted control transfer instruction;

- Edge Case Validation: In our final phase of testing, we focused on edge case validation. This involved a series of comprehensive tests designed to assess the system's behavior in unusual or extreme scenarios. Specifically, we conducted experiments to observe how the system reacts when the shadow stack is either empty or full. These conditions were intentionally engineered to push the limits of the system, allowing us to identify potential vulnerabilities and ensure robust performance under such peculiar circumstances. Through these tests, we aimed to gain a deeper understanding of the system's stability and reliability when faced with atypical operational situations.

The project has undergone rigorous testing across a range of potential attack scenarios and unforeseen circumstances. The results of these tests indicate that the system demonstrates a robust ability to handle these challenges effectively. This reliability ensures that the Control Flow Integrity of the code remains intact, instilling confidence that the system can be trusted to function securely in various threat environments.

6.2 Limitations

While we introduce robust protections against control flow hijacking attacks, certain limitations must be acknowledged.

Firstly, the project lacks coverage against non-control-flow hijacking attacks. The system does not mitigate side-channel attacks, such as timing or power analysis, which require separate countermeasures. Data-oriented attacks that do not rely on altering the control flow are also outside the scope of this implementation. While this does not represent a problem for the project development itself as these threats were out of scope during the definition part of the project, as described in the threat model, they are indeed a real-life problem. This means that the project must be paired with other security measures to effectively protect a device from various types of attacks.

Moreover, we assume software integrity to be guaranteed. The system assumes that the source code of the project, including the compilers, is trusted and free of pre-existing vulnerabilities. Compromised source code could undermine the integrity of the Control Flow Integrity enforcer protections. Furthermore, if the source code gets compromised, we can't ensure its correctness in protecting the Control Flow Integrity of the code. Thus, additional security measures must be put in place to avoid possible tampering with the code. Again, as already explained in the threat model, this is out of scope as we assume that the code is inherently safe.

Additionally, there could be edge cases that have not been discovered, which could lead to potential exploitations of the Control Flow Integrity enforcer. However, these can be discovered through exhaustive testing and will be part of future development.

Lastly, we can't know if vulnerabilities related to the *RISC-V* Instruction Set Architecture could lead to the exploitation of the user code. In such a case, we can't be sure that the project will be able to protect the Control Flow Integrity and prevent exploitation. For example, the recently discovered hardware vulnerability in *T-Head's XuanTie C910* and *C920* CPUs named *GhostWrite* (Fabian Thomas, Lorenz Hetterich et al., 2024)[6] allows an attacker with limited privileges to read and write from and to physical memory. This type of vulnerability could not be prevented with the provided security features as a threat actor may be able to overwrite Physical Memory Protection configurations or the shadow stack and Control Flow Graph themselves to modify the stored data, leading to a compromisation of the Control Flow Integrity enforcer.

In summary, we provide robust protection against unauthorized control transfers in *RISC-V* microcontrollers, particularly effective in embedded environments where code integrity is paramount.

[2] Note that this is an example as the compiler usually uses register *a5* to store the address of indirect jump instructions by calling convention.

While the project relies on PMP for data protection, it nonetheless establishes a secure foundation for Control Flow Integrity enforcement, with the potential for further optimization in future iterations. Moreover, other security mechanisms could be implemented to completely secure an embedded device.

7 Performance Analysis

In this chapter, we examine and discuss the project's performance impact on execution time and memory usage. Firstly, we present data regarding the time overhead imposed by the project during execution. During the time analysis, we describe the meaning of the collected data and the variations from our expectations. Secondly, we present the memory footprint that the Control Flow Integrity enforcer has on the produced binary. Furthermore, we provide an analysis of the obtained results as well as a comparison with results obtained in similar projects. Lastly, we provide some optimization ideas to further reduce the performance impact on the code.

During the performance analysis, we tried to perform tests similar to the one proposed in the articles *Efficient CFI Enforcement for Embedded Systems Using ARM TrustZone-M* (Gisu Yeo, Yeryeong Kim et al., 2022)[9] and *PROLEPSIS: Binary Analysis and Instrumentation of IoT Software for Control-Flow Integrity* (Valentina Forte, Nicolò Maunero et al., 2021)[20] where the writers propose similar solutions to enforce Control Flow Integrity on *ARM*-based embedded devices. We tested the infrastructure on the same algorithms proposed by the papers, resulting in tests on 13 different algorithms plus 4 variations with "forced" indirect jumps[1].

Note that, to provide realistic data, the aforementioned *Espressif's ESP32-C3-DevKitM-1* has been used to conduct the tests.

7.1 Time Overhead

In this section, we discuss how the infrastructure affects the execution time of the code. Two bar charts depicting test results can be seen in Figures 7.1 and 7.2, note that execution times have been separated into two different charts to enhance the readability of data.

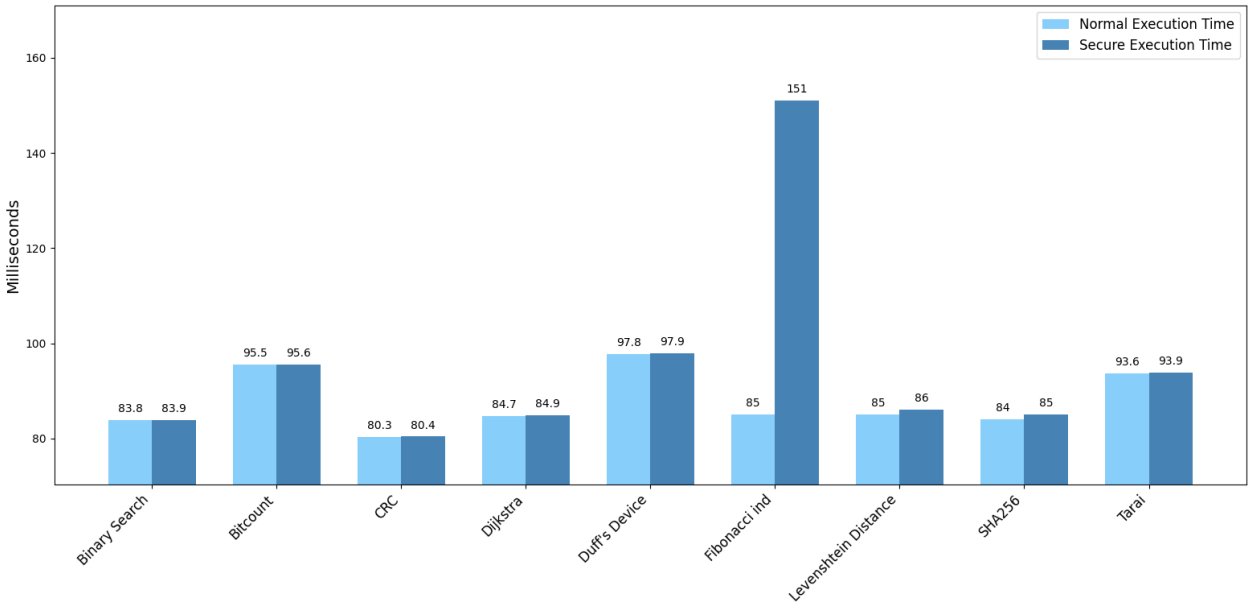


Figure 7.1: Comparison between execution times (low times)

Table 7.1 depicts test results for each algorithm as well as the percentage of time overhead. In most cases, execution time increases by a few milliseconds or less, resulting in an unnoticeable impact on execution time. However, there are two cases in which the time overhead impacts performances in

[1] Indirect jumps were "forced" using volatile function pointers of the *C* programming language.

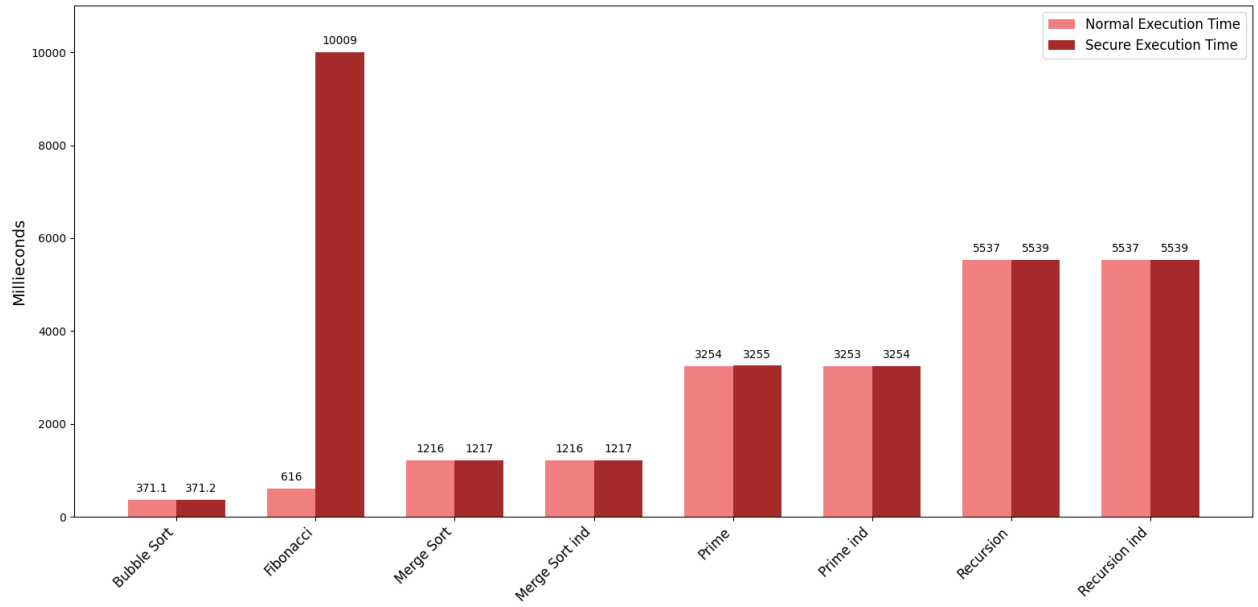


Figure 7.2: Comparison between execution times (medium and high times)

a relevant way.

It is straightforward to understand why the overhead is small for algorithms like *Binary Search* or *Duff's Device* since these programs perform few jump instructions and most of the computation is performed inside a while or for loop so, the overhead given by the instrumentation is unnoticeable if compared to the total execution time. For example, the *Binary Search* algorithm performs 3 jump instruction and 3 return instructions, even if it is working on a rather big input array. This means that, despite the size of the input, the Control Flow Integrity enforcer is called only 6 times, resulting in a small time overhead. We want to specify that this behavior is expected in these situations.

Instead, examples like *Prime* and *Recursive* are a bit trickier. Since these algorithms work with a recursive approach, they perform many control transfer instructions consequently, thus we expect the time required for the context switch and the forward and backward edge controls to be somewhat impactful. However, results show that even in these cases, the difference is barely noticeable[2]. The reason could be due to the processor's frequency, which, in the case of *Espressif's ESP32-C3-DevKitM-1*, is 160 MHz, meaning that the CPU is able to execute 160 million operations in one second. If we take the *Recursion* algorithm as an example, we see that it performs $\sim$ 4000 jump instructions and $\sim$ 4000 return instructions which translates to $\sim$ 8000 edge controls and $\sim$ 16000 context switches[3]. However, if we compare those numbers to the 160 million operations the processor can perform in a second, we see why the time overhead is so small. Note that we inject 2 instructions before every jump and 3 instructions before every return, resulting in an instruction overhead of $(4000 * 2) + (4000 * 3) = 20000$. Moreover, the handling of forward and backward edge controls requires $\sim$ 180 instructions, which results in $(4000 + 4000) * 180 = 1440000$ total instructions. If we sum up the values, we obtain a total instruction overhead of $20000 + 1440000 = 1460000$. Now, if we compute $\frac{1460000}{160000000}$ we discover that the utilized processor can perform all these instructions in $\sim$ 9.125 milliseconds. Depending on the algorithm and the recursion depth, we see that the overhead affects the execution time by a few milliseconds, thus matching the results obtained during testing.

Two peculiar cases are the ones of the *Fibonacci* and *Fibonacci Indirect* algorithms, in these cases, the overheads are 1523.24% and 77.97%, respectively. The possible reason for these enormous overheads is that these algorithms make two recursive calls each time, so we end up with an exponential

[2]In these cases, we see that the execution time increases by a few milliseconds instead of by less than a millisecond.

[3]The number is doubled because, for each edge control, we need to increase the privilege to M-mode, perform the control, and then return to U-mode, resulting in two context switches.

Algorithm	Normal run time (ms)	Secure run time (ms)	Time Overhead
Binary Search	83.87	83.88	0.012%
Bitcount	95.5	95.6	0.11%
Bubble Sort	371.13	371.16	0.008%
CRC	80.25	80.26	0.012%
Dijkstra	84.7	84.9	0.236%
Duff's Device	97.795	97.799	0.0041%
Fibonacci	616.6	10008.9	1523.24%
Fibonacci Indirect	84.9	151.1	77.97%
Levenshtein Distance	85	86	1.176%
Merge Sort	1216	1217	0.0822%
Merge Sort Indirect	1216.8	1216.9	0.0082%
Prime	3254	3255	0.03%
Prime Indirect	3253	3254	0.03%
Recursion	5537	5539	0.036%
Recursion Indirect	5537	5539	0.036%
SHA256	84	85	1.19%
Tarai	93.6	93.9	0.32%

Table 7.1: Test results for execution times

amount of jump and return instructions. This leads to many invocations of the Control Flow Integrity enforcer, thus the execution time is strongly affected by this.

Overall, we achieved an average time overhead of 94.38% with a standard deviation of 368.68. However, if we do not take into account outliers, we can see that the median is 0.036%. These results clearly show that the time overhead is almost unnoticeable in many cases but, this also demonstrates that the project strongly suffers from deeply recursive algorithms and that further optimization is required to reduce the time impact on these types of algorithms.

Lastly, in Table 7.2, we provide the times required to instrument the code, extract the Control Flow Graph, and perform the simulation. The instrumentation and CFG extraction phases always require less than a fraction of a second to finish. However, we must take into consideration that the tested algorithms are composed of few lines of code and the situation could change if we were to instrument a firmware composed of thousands of lines. The simulation instead requires a lot of time, even algorithms that run in a second or less require some seconds to simulate. This is due to the logging required to extract the indirect jump destinations, which introduces a noticeable overhead. Note that we can accept this result since the simulation is run only once and the logging is removed from the final binary so that this overhead does not affect the actual execution time.

7.2 Memory Overhead

In this section, we showcase how the infrastructure affects the size of the produced binary. A bar chart depicting test results can be seen in Figure 7.3.

Table 7.3 depicts test results for each algorithm as well as the percentage of memory overhead. As we can see, in most cases, the size of the binary increases by $\sim 10\%$. This is due to the shadow stack, Control Flow Graph, and the instructions added during instrumentation.

As for the time overhead analysis, there are some peculiar cases. *Prime*, *Recursion* and their indirect variations *Prime Indirect* and *Recursion Indirect*, present a memory overhead of $\sim 200\%$ and $\sim 450\%$ respectively. This happens because these recursive algorithms perform all the jump instructions consecutively and then all the return instructions. This means that we must allocate a shadow stack big enough to store all the return addresses to ensure correct backward edge controls. For example, we have seen that the *Recursion* algorithm performs ~ 4000 jump instructions, which means that we must create a shadow stack able to store at least 4000 addresses. Given that an address is stored as an *unsigned int*, we can easily see that a shadow stack that can store 4000 addresses occupies

Algorithm	Instrumentation (ms)	Simulation (ms)	CFG extraction (ms)
Binary Search	2.96	no simulation	1.74
Bitcount	1.96	no simulation	1.53
Bubble Sort	3.02	no simulation	1.75
CRC	2.03	no simulation	1.54
Dijkstra	2.64	no simulation	1.82
Duff's Device	2.65	no simulation	1.63
Fibonacci	2.09	no simulation	1.49
Fibonacci Indirect	2.02	130556.7	70.37
Levenshtein Distance	2.55	no simulation	1.67
Merge Sort	3.54	no simulation	1.78
Merge Sort Indirect	3.6	4639.4	2.73
Prime	2.14	no simulation	1.68
Prime Indirect	2.08	11405.1	4.51
Recursion	2.02	no simulation	1.59
Recursion Indirect	2.02	23275.1	8.75
SHA256	4.11	no simulation	1.89
Tarai	1.98	no simulation	1.52

Table 7.2: Test results for instrumentation, simulation, and CFG extraction phases

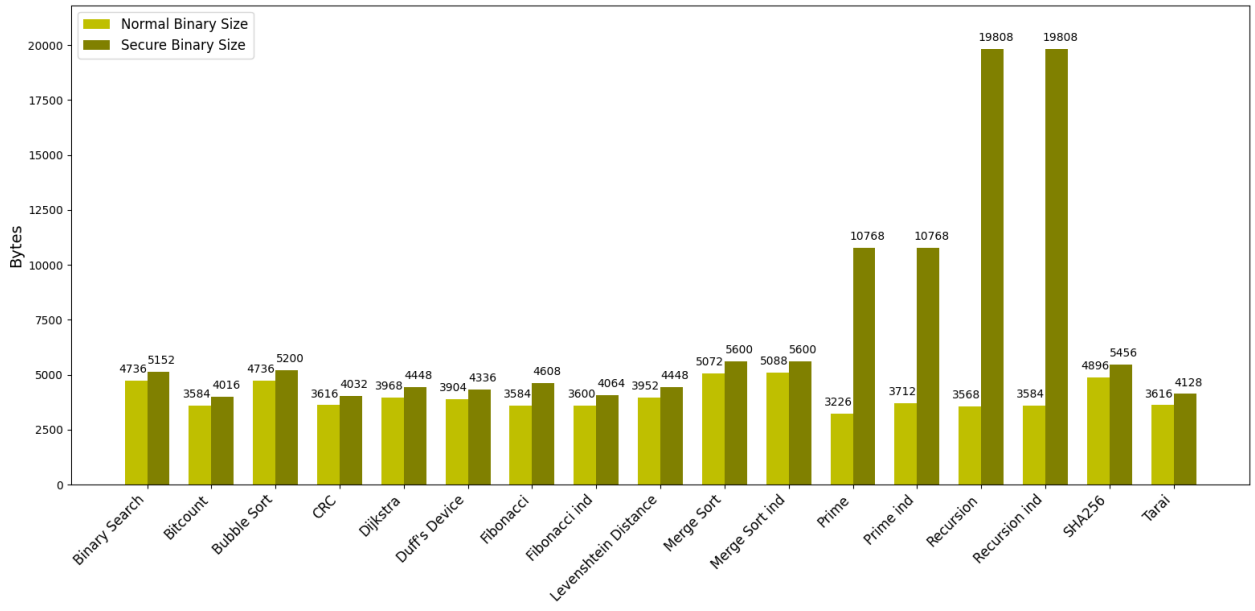


Figure 7.3: Comparison between binaries size

Algorithm	Normal bin size (B)	Secure bin size (B)	Memory Overhead
Binary Search	4736	5152	8.78%
Bitcount	3584	4016	12.05%
Bubble Sort	4736	5200	9.79%
CRC	3616	4032	11.51%
Dijkstra	3968	4448	12.09%
Duff's Device	3904	4336	11.06%
Fibonacci	3584	4608	28.57%
Fibonacci Indirect	3600	4064	12.88%
Levenshtein Distance	3952	4448	12.55%
Merge Sort	5072	5600	10.41%
Merge Sort Indirect	5088	5600	10.06%
Prime	3226	10768	233.78%
Prime Indirect	3712	10768	190.08%
Recursion	3568	19808	455.15%
Recursion Indirect	3584	19808	452.68%
SHA256	4896	5456	11.43%
Tarai	3616	4128	14.16%

Table 7.3: Test results for memory consumption

$4000 * 4 = 16000$ *Bytes*[4]. Now, if we take the size of the secure binary for the *Recursion* algorithm, we can calculate $19808 - 16000 = 3808$ *Bytes*, so, apart from the shadow stack, the binary has a memory overhead of $\frac{3808-3568}{3568} * 100 =\sim 6.73\%$ which is similar to other ones.

Note that this problem is related strictly to recursive algorithms as they perform a high number of jumps and then they start performing the first return instruction. However, we could have a similar problem with the Control Flow Graph as for big binaries like firmwares, we can have many different jump instructions, increasing the CFG'size.

Overall, we achieved an average memory overhead of 88.06% with a standard deviation of 152.77. However, similarly to the time overhead, if we do not take into consideration outliers, we can see that the median is 12.09%. These results show that, in most cases, the project can produce a secure binary without affecting too much memory consumption. Even in this case, the project suffers from deeply recursive algorithms as we need to allocate a bigger shadow stack to accommodate space for secure storage of each return address.

7.3 Results and Analysis

In this section, we discuss the obtained result and showcase the weak points of the implementation.

As we have seen in the previous sections, the project showcases promising performances in most cases. In almost all the tested algorithms, time overhead was barely noticeable and memory consumption was acceptable. Overall, the project infrastructure does not impact the performance of an executable.

However, in the case of deeply recursive algorithms, we have seen that the time overhead grows exponentially due to the high number of control transfer instructions and the related forward and backward edge controls. Also, the size of the binary grows as we need a bigger shadow stack to store all the addresses needed to perform the backward edge controls.

Moreover, we have seen that if the code is composed of thousands of lines, it is likely that the Control Flow Graph will be very big. This is because if we have many jump instructions from different sources to different destinations, we must store each pair, thus increasing the space occupied by the CFG.

Similar results have been obtained in the already cited paper *PROLEPSIS: Binary Analysis and Instrumentation of IoT Software for Control-Flow Integrity*[20] where the writers introduced an average

[4]We multiply by 4 because it is the size of an *unsigned int* in C.

instruction overhead of $\sim 4\%$[5]. Unfortunately, given the fact that the writers proposed a preliminary prototype, they do not provide the execution time overhead introduced with their technology. Moreover, the paper *Efficient CFI Enforcement for Embedded Systems Using ARM TrustZone-M* [9] presents an average execution time overhead of 159.3%, with the lowest overhead being 0.2% and the highest being 652.27%. Additionally, the writers say that they saw an average 7.36% increase in memory consumption. With these results, we can effectively say that the proposed solution to enforce Control Flow Integrity achieved standard overheads in both execution time and memory consumption, making it a valuable choice for embedded systems.

In the following section, we provide some optimization techniques that could be applied to further increase the performance. Specifically, we provide alternative solutions to improve the project's weak points.

7.4 Optimization Techniques

Since optimization is a key factor when developing projects on embedded devices, given their hardware limitations, we tried to provide an infrastructure as fast as possible while preserving memory usage. However, as test results show, there are some aspects of the project that require further improvement to be acceptable.

The two identified cases in which the project strongly impacts the performances are:

- Deeply recursive algorithms: we have seen that deeply recursive algorithms require a big shadow stack to hold all the return addresses, thus they require a lot of extra memory. To address this issue, we could modify the shadow stack in the following way. When we see that a function calls itself many times, we could store the return address only one time and introduce a *peek* function. With this, we could still perform the backward edge controls when we need to, but we would need to store the address only once. Then, if we need to perform a control on another address, we would just need to *pop* the "recursive" address and then *pop* the following one to see if it matches with the one we are checking. With such modification, we could effectively reduce the amount of space occupied by the shadow stack in case of recursive algorithms without impacting the provided security features;

- Big user code: we have seen that with a very big codebase (like the aforementioned firmware one) we could end up in a situation where the Control Flow Graph grows bigger and bigger due to the high amount of control transfer instructions from different sources to different destinations. Unfortunately, we can do nothing about memory consumption in this case, as we can't afford to avoid storing some source-destination pairs. However, as already explained we could implement a Hash Table, which would not decrease the memory consumption but, at least, we could reduce the time required to access the Control Flow Graph from $\mathcal{O}(\log n)$ to $\mathcal{O}(1)$.

Lastly, we provide a space optimization solution that could be implemented for very small source code. Say, for example, that the *.text* section that we want to instrument starts at address $0x4038A000$ and ends at address $0x4038F000$. Since the first part of the address stays the same it provides no useful information, thus we could apply a bit-mask to remove 4038 from the address, storing only the last 16 bits. This means that we would be able to store an address as an *unsigned short* instead of an *unsigned int*. As a result, we would be able to halve the space required to store the Control Flow Graph and the shadow stack, resulting in a great improvement in memory consumption. However, this solution only works when the user code is very small because if the *.text* section starts at address $0x4038A000$ and ends at address $0x4039F000$, we would still need an *unsigned int* to store a correct representation of the address.

In conclusion, this analysis has demonstrated that our project exhibits commendable performance in a variety of scenarios. Nevertheless, there are specific edge cases where its efficiency can be significantly compromised, leading to suboptimal results. These instances reveal potential vulnerabilities in the system that could hinder its overall effectiveness. To address these challenges, we suggest implementing the proposed solutions, which aim to refine the project's functionality. By fine-tuning the

[5]Note that the writers provided the instruction overhead and not the total binary size overhead.

system to better adapt to particular situations, we can effectively minimize both memory usage and execution time, thereby enhancing the overall performance and reliability.

8 Real-Time Operating Systems

This project has been developed and tested on bare-metal applications to ensure its correctness in simple environments. However, many embedded devices are designed to accomplish more complex tasks and they need special environments to support the execution. The purpose of this chapter is to provide insights into how the project could be implemented to work with Real-Time Operating Systems (RTOS) to secure and support complex tasks and environments. Specifically, we provide an implementation idea for two famous RTOSes, *Free Real-Time Operating System* (*FreeRTOS*) and *Zephyr RTOS*, depicting their strengths and the limitations that may arise during implementation.

8.1 FreeRTOS

FreeRTOS[15] is an open-source, lightweight, Real-Time Operating System kernel designed for embedded systems. In recent years, *FreeRTOS* has become one of the most widely used *RTOS* kernels in embedded applications, particularly in microcontrollers, because of its efficiency, scalability, and portability. *FreeRTOS* is mostly used in applications that require reliable, deterministic behavior in resource-constrained environments such as medical devices, IoT devices, and automotive systems.

Among the many key features provided by *FreeRTOS*, there are:

- Real-Time Scheduling: *FreeRTOS* supports real-time scheduling with preemptive and cooperative multitasking. In preemptive scheduling, tasks are prioritized and can interrupt lower-priority tasks, while in cooperative multitasking, tasks yield control to others explicitly. Moreover, it uses priority-based scheduling, allowing developers to assign priority levels to tasks based on their timing requirements;

- Task Management: Tasks in *FreeRTOS* are individual threads of execution, each with its stack and context. The kernel allows the creation, deletion, and management of tasks dynamically. Each task operates in its context, which the kernel saves and restores during context switching;

- Memory Management: *FreeRTOS* provides several memory allocation schemes, including static and dynamic allocation, through its portable memory allocation subsystem. It supports different heap management schemes, offering varying levels of complexity and memory usage optimization. These schemes allow developers to balance between simplicity and flexibility;

- Interrupt Handling: *FreeRTOS* is designed to work seamlessly with hardware interrupts, allowing Interrupt Service Routines (ISR) to communicate with tasks through mechanisms like queues and semaphores. The kernel offers a low-latency method for ISR handling, enabling efficient communication between ISRs and tasks while ensuring minimal delay in task scheduling;

- Software Timers: *FreeRTOS* includes a timer API, allowing developers to create software timers that automatically trigger callback functions after a specified period. This feature helps to manage time-dependent operations without creating dedicated tasks;

- Portability and Scalability: *FreeRTOS* is highly portable and can be adapted to run on numerous processor architectures, including *ARM Cortex-M*, *ARM Cortex-A*, *RISC-V*, and many others. It is structured in a way that allows developers to scale applications easily by adding or removing tasks, memory management schemes, and communication mechanisms as needed.

With all these features it is easy to see why *FreeRTOS* has become a standard in the community. However, *FreeRTOS* presents some limitations like the limited built-in security and the lack of support for multicore processing. Some of these flaws have been addressed by community users and companies with alternative versions of the Operating System. For example, *Espressif* presented *IDF*

FreeRTOS[5], a version of *FreeRTOS* which provides support for multicore processing. Moreover, *high integrity systems* developed *SAFERTOS*[10], a redesign of the *FreeRTOS* kernel to enhance security.

Overall, *FreeRTOS* is a powerful choice for developers working with embedded systems that require real-time capabilities, low overhead, and efficient resource management. Its design balances simplicity with flexibility, making it suitable for a wide variety of applications from small IoT devices to more complex industrial systems.

8.2 Zephyr RTOS

Zephyr RTOS[22] is an open-source, Real-Time Operating System designed for embedded systems and IoT applications. Developed under the Linux Foundation, it offers a lightweight, modular, and scalable platform that supports a wide range of devices, from simple microcontrollers to complex systems. Widely adopted across IoT devices, industrial automation, medical equipment, and automotive systems, *Zephyr RTOS* excels in environments requiring reliability, efficiency, and resource optimization.

Zephyr is extremely scalable and modular. Its architecture enables developers to include only the features they need, optimizing memory usage and performance for resource-constrained devices. The kernel supports multiple configurations, making it adaptable for both simple single-threaded tasks and advanced multi-threaded applications. Additionally, *Zephyr* is designed to deliver deterministic, time-critical performance, ensuring minimal latency and high reliability.

Zephyr includes advanced security features such as memory protection, secure boot, and support for Hardware Security Modules (HSMs). Regular updates and audits ensure compliance with stringent security standards, making it a robust choice for applications demanding high reliability and safety.

While both *Zephyr RTOS* and *FreeRTOS* are widely used in embedded systems, they differ in their architecture, feature set, and target use cases. Firstly, *Zephyr* provides a comprehensive ecosystem out of the box, including networking stacks, device drivers, and file systems. *FreeRTOS*, in contrast, offers a minimal kernel focused on task scheduling and real-time performance, with additional features like networking provided as optional libraries. Moreover, *Zephyr* supports more complex applications, including those requiring multi-threading and multicore systems, making it suitable for advanced IoT and industrial devices, while *FreeRTOS* is lightweight and optimized for simple real-time tasks on resource-constrained microcontrollers. Lastly, *Zephyr* offers built-in security features while *FreeRTOS* provides basic security features and relies on external libraries for advanced functionalities.

Overall, with its lightweight design, real-time capabilities, and extensive ecosystem, *Zephyr RTOS* is a versatile platform for developers building secure, scalable, and efficient embedded solutions.

8.3 FreeRTOS and Zephyr RTOS Implementation

In this section, we provide a detailed description of how *FreeRTOS* could be implemented to work inside the provided architecture and why this could increase the scenarios in which it could be used. Note that we focus on the implementation of *FreeRTOS* but, most of the information provided in this section can be applied to the implementation of *Zephyr RTOS*. The same is true for the limitations discussed in section 8.4 as the RTOSes suffer from the same implementation problem.

We have seen that the project is designed to enforce Control Flow Integrity on running untrusted user code in bare-metal environments. However, given its simplicity, it could be used to secure code in more complex environments, for example, with a Real-Time Operating System like *FreeRTOS*.

The main idea is to maintain the Control Flow Integrity enforcer as the M-mode operator, managing system boot and edge controls. The user code, in this case, is represented by the various tasks generated thanks to *FreeRTOS*. Code instrumentation would not change much as we would just need to add the *regex* functions to search for *FreeRTOS*-specific instructions that cause a control transfer during execution and inject the relative instructions. Finally, *FreeRTOS* could be used as a supervisor, either trusted or untrusted, depending on the case. It would be in charge of managing task scheduling and memory operations.

Moreover, since *FreeRTOS* requires its interrupt service routine, we could either:

- Prepare two separate interrupt vector tables, one for the project[1] and one for *FreeRTOS*. However, traps at the user level are managed by default at the machine level so, if we have an interrupt or an exception in one of the tasks, the control is transferred to the project's interrupt vector table, and we must manually transfer the flow to the correct handler of the *FreeRTOS* interrupt vector table. A solution to this could be to prepare two separate interrupt vector tables and use *mideleg* and *medeleg* registers to automatically delegate some interrupts and/or exceptions to the *FreeRTOS* interrupt vector table. For example, if we know that user interrupts are always managed by *FreeRTOS* we could insert the corresponding code inside *mideleg*. With this, each time a user-level interrupt is generated it is trapped at the *FreeRTOS* interrupt vector table. Note that this does not mean that the Control Flow Integrity enforcer will not be able to manage those traps as higher-level interrupts and exceptions can't be delegated to the lower-level handler. For example, if a machine-level interrupt is generated, it will always be handled by the privileged interrupt vector table;

- Another solution would be to use only the project's interrupt vector table, specifying that *FreeRTOS* should trap exceptions and interrupts at such table. This means that every time a trap is taken in *FreeRTOS* the execution is transferred to the project's handlers. However, this may result in two unwanted situations. Firstly, handling all the traps inside the project's interrupt vector table may require many controls, and the code may become overly verbose. Secondly, this means that the traps that are generated in *FreeRTOS* are handled in machine mode, and, for some environments, this could constitute a problem if we want to keep separation between privileges.

In summary, the integration of *FreeRTOS* or *Zephyr RTOS* into the proposed architecture demonstrates a promising pathway to expand the system's applicability, enabling robust real-time task management while maintaining critical security assurances through Control Flow Integrity enforcement.

8.4 Porting Limitations

The proposed implementation suffers from one key limitation, *FreeRTOS* needs to perform some operations on Control and Status Registers to function correctly but, as we have explained, such registers are only available in machine mode. Since the implementation would require *FreeRTOS* to run either in supervisor or user mode, all those operations would generate an illegal instruction exception as those privilege modes can't access machine CSRs. The critical operations on CSRs are depicted in Listing 8.1.

To address this problem, we could enhance the instrumentation phase to modify *FreeRTOS*'s code in the following way. Firstly, we need to find the critical instructions, which can be done easily with *regex* functions. An example of a *regex* function to locate critical instructions is *(csrr[wcs])\s*([a-z0-9]+),\s*([a-z0-9]+)*. With this, we can extract the operation that *FreeRTOS* is trying to perform (among the one seen in section 4.7.), the target CSR, and the value that is being written. Then, the target CSR can be translated thanks to the table *Currently allocated RISC-V machine-level CSR addresses*[2] where we can see that for example register *mstatus* is stored at address $0x300$. Now we can precisely describe each target CSR, and we just need to inject new instructions to perform an *ECALL* to the project's interrupt vector table, which will perform the modification of the CSRs on behalf of *FreeRTOS*.

To do so, we can choose between two options. The first is to use register *a7* to hold the new *ECALL* code which determines the operation and three other registers, say *a4*, *a5*, and *a6* to hold the CSR address, the CSR instruction, and the value being written respectively. However, this solution is highly inefficient in the use of registers since we could encode those values into 3 registers instead of 4. Alternatively, we could use register *a7* to hold the new *ECALL* code, register *a6* to hold the CSR address and the CSR instruction, and register *a5* to hold the value in case of writing or setting operations. Since there are 4 CSR instructions we just need 2 bits to encode them, Table 8.1 depicts a possible representation of the encodings of CSR instructions. All other bits would be

[1] The same we have seen in section 5.5.

[2] Provided by the *RISC-V Privileged Manual*[16] at page 17.

Number	Value	Instruction	Use
1	00	CSRRW	Write the target CSR
2	01	CSRRS	Set bit(s) in the target CSR
3	10	CSRRC	Clear bit(s) in the target CSR
4	11	CSRR	Read the target CSR

Table 8.1: CSR instructions encoding for *FreeRTOS*

reserved to describe the target CSR for the operation. Note that CSRs addresses go from $0x300$ up to $0xF15$ so, we need a maximum of 12 bits to represent each of them. Also, we can't encode the *ECALL* code inside the same register because we are using the least significant bit to represent forward and backward edge controls, thus using it would result in ambiguity on the operation to perform.

```
1  csrrw mstatus, reg
2  csrrw mepc, reg
3  csrrw mtvec, reg
4  csrrw mie, reg
5  csrrw mtval, reg
6  csrrw mip, reg
7  csrrw mscratch, reg
8  csrrw medeleg, reg
9  csrrw mideleg, reg
10
11 csrr reg, mstatus
12 csrr reg, mepc
13 csrr reg, mtvec
14 csrr reg, mie
15 csrr reg, mtval
16 csrr reg, mip
17 csrr reg, mhartid
18 csrr reg, mcause
19 csrr reg, misa
20
21 csrrc mstatus, reg
22 csrrc medeleg, reg
23
24 csrrs mstatus, reg
25 csrrs mie, reg
```

Listing 8.1: *FreeRTOS* operations on Control and Status Registers

Listing 8.2 depicts the instrumentation needed to correctly delegate the privileged instructions to M-mode. Say that we need to perform the instruction *csrrw mstatus, a0*, we know that *mstatus* is represented by $0x300$ and that the operation *csrrw* is encoded as 00. Firstly, we load the value we want to write, which is stored in register *a0* into register *a5*. Secondly, we load the address of the CSR into register *a6*. After that, we use an *ori* instruction to set the most significant bits depending on the CSR operation. For example, *csrrc* is encoded as 11 so, we need to perform the *or* operation with the value $0xC0000000$. Lastly, we load into register *a7* the *ECALL* code that we want to use to represent these types of calls. Once all the values are loaded, we can perform the *ECALL* instruction itself, the control will be transferred to the correct handler, and the operation will be performed on behalf of *FreeRTOS*. As soon as the operation is handled, execution is resumed at the instruction that was interrupted. Note that in the case of *csrr* operations, we would need to use a register to hold the value of the CSR. In this case, the privileged handler would read the CSR and store its content inside register *a5* and *FreeRTOS* would use such register to access the content of the CSR after the

ECALL instruction. Figure 8.1 a depicts a possible encoding for register *a6* where we use the two most significant bits to encode the CSR instruction we want to perform and all the other bits to represent the target CSR.

Figure 8.1: Possible encoding for register *a6*

Inside the interrupt vector table, we would just need to add an if statement to determine if we are requesting to perform a CSR instruction on behalf of *FreeRTOS* based on the *ECALL* code.

```
mv a5, w_value      # Move the source value into register a5
la a6, csr          # Load the address of the target CSR
ori a6, a6, csr_op  # Or with the encoding of the CSR operation
li a7, ecode        # Load the new ECALL code
ecall               # Perform the ECALL instruction
```

Listing 8.2: *FreeRTOS* instrumentation for Control and Status Register operations

Note that this is just a proposal of implementation, and there are other ways to implement *FreeRTOS* within the project. For example, instead of encoding the CSR instruction inside register *a6*, we could define an *ECALL* code for each operation and simply use that value to determine the operation to perform. This would result in fewer instructions to inject (as we could remove the *ori* instruction) but would increase the size of the handler as we would need to provide controls for each implemented *ecode*.

9 Future Works

In this thesis, we presented a Control Flow Integrity enforcer that poses itself as a foundational approach to improving the security of *RISC-V*-based embedded systems. Moreover, the project showed promising results in the security and performance analysis, making it a suitable choice for real-world applications. However, further advancements can enhance its applicability, efficiency, and robustness. This chapter outlines potential directions for future work to expand the project's capabilities.

9.1 Additional Security Mechanisms

In previous chapters, we discussed and demonstrated the ability of the infrastructure to detect and prevent control flow hijacking attacks, such as *Return-Oriented Programming* or *Jump-Oriented Programming*. However, in security-demanding environments, it may be necessary to introduce further techniques to protect the Control Flow Integrity of the code. Here, we list some security mechanisms we plan to integrate with future updates.

In section 2.2.3, we introduced the *Stack Canary*, a security mechanism used to detect and prevent stack-based buffer overflow attacks, a common type of vulnerability in programs. It involves placing a small, random value in memory right before the stack's return address. This value acts as a sentinel and is checked for integrity before the function returns control to its caller. The *Stack Canary* is simple yet effective as it provides a straightforward way to detect stack corruption caused by buffer overflows. Overall, we think that introducing a *Stack Canary* in the project would be a good addition to the security features as it would provide a slight increase in security for environments that require a high degree of protection.

Since the project suffers from source code modification, as we have no way of telling if the code was modified or not, and this could affect the correctness of forward and backward edge controls, we could add a hashing function to the project. With this function, we could generate a hash of the source code. Note that we plan to take into account only the "static" part of the code and not the code imported by the user. With this, we could perform a check at each compilation by simply comparing the initial hash with the freshly generated one. If the hash differs, it means that the source code has been tampered with and should not be trusted. This would lead to a simple and fast way to determine if the source code is secure or if it has been modified.

Moreover, many device-specific security measures could be explored as there may be some devices that provide hardware modules that could be used to increase the security of the project. However, this process requires exhaustive testing and would lead to possible enhancement only on a few embedded devices.

Note that we can't add security measures such as Data Execution Prevention or Address Space Layout Randomization to the binary as they would conflict with the system's configuration. Data Execution Prevention requires setting memory regions with either write but not execute privileges or execute but not write privileges. This would impact the project as it provides memory regions that require both privileges. Address Space Layout Randomization instead would randomize all the addresses so that they are different at each execution. While this technique provides good security, it would render the Control Flow Graph completely useless as the addresses we collect during the extraction would be different from the addresses at execution time, leading to a situation where each forward edge control fails.

Lastly, note that the project is designed to protect the code from control flow hijacking attacks and must be paired with other security measures before deployment. This is needed to ensure security on every attack surface. Otherwise, a threat actor may be able to exploit other vulnerabilities to perpetrate an attack bypassing the provided security features.

9.2 Code Optimization

Although the project has been designed with performance considerations, further optimizations could enhance its usability in a broader range of embedded applications. During the performance analysis in chapter 7, we have seen that the system comes with an acceptable average time and memory overhead. However, there are edge cases where this is false, and the overhead grows exponentially. In future works, we plan to provide optimization solutions to cover such cases, lowering the time and space impact on the execution.

Firstly, we propose a solution to address the space overhead of deeply recursive algorithms. We have seen that in these cases, we need to allocate a very big shadow stack, which increases the size of the produced binary. To solve this, we propose to add a *peek* function, which allows us to look at the first element of the shadow stack without removing it. When we need to perform a forward edge control and consequently push the return address into the shadow stack, we first check the top value. If the addresses are the same, we avoid pushing the same value more times. On the other hand, when we need to perform a backward edge control, we look at the first value of the shadow stack. If the two addresses are equal, we approve the return instruction without popping the value. Instead, if the addresses differ, we pop two values and compare the second one with the return address we are checking. With this solution, we could effectively address the memory consumption problem of deeply recursive algorithms without affecting the security capabilities of the project.

Secondly, we offered a solution to decrease memory usage when the user code is minimal. For example, say that the user code starts at address $0x4038A000$ and ends at $0x4038F000$. In this case, we can ignore the first 16 bits of the address as they do not provide useful information. This means that we could modify the shadow stack and Control Flow Graph to store *unsigned short* values instead of *unsigned int* values. Such a solution would halve the memory requirement for the shadow stack and CFG. Although this solution is very effective in memory management, it can only be applied when the codebase is very small.

Moreover, we have seen that if the user-imported code is very big, take as an example a firmware, the Control Flow Graph would drastically increase in size. This is because the larger the code, the larger the possibility of having many jump instructions from different source addresses to different destination addresses. To address this problem, we propose to use a Hash Map instead of a two-dimensional array to represent the CFG. Although this solution does not reduce memory usage, it allows for faster lookups as Hash Maps provide accesses in constant time ($\mathcal{O}(1)$).

Lastly, translating the source code from the *C* programming language into pure *Assembly* language can significantly enhance performance. This conversion allows for more fine-grained control over system resources, enabling writing code tailored specifically to the hardware on which it runs. As a result, it opens up opportunities for advanced optimization techniques that can minimize time and memory overhead. By leveraging the specific architecture of the processor, programmers can streamline execution paths, reduce unnecessary computations, and manage memory usage more efficiently, which collectively contributes to a more responsive and efficient application.

9.3 Exhaustive Testing

During the security analysis in chapter 6, we showcased the effectiveness of the project in protecting the Control Flow Integrity of the user code. Moreover, we proved through tests that attempts to perform an unauthorized control transfer are immediately detected and prevented by the Control Flow Integrity enforcer. However, we also pointed out that the presented project is not able to mitigate non-control-flow hijacking attacks. In future works, we plan to perform exhaustive testing of the project to further inspect its behavior in diverse environments. This could lead to either stronger proof that the project is actually able to protect the code or to the discovery of untested scenarios. In the latter, we plan to provide a solution to eventual unexpected behavior.

9.4 RTOS Implementation

The main focus for future works sees the integration of a Real-Time Operating System such as *FreeRTOS* or *Zephyr RTOS*. Such a task is of great importance since an implementation providing the

features of both the project and an RTOS would drastically increase the number of fields in which this project could be deployed. This would open the door to more complex solutions and allow embedded devices to carry out complex tasks while the project enforces Control Flow Integrity.

We already discussed all the additional features that an RTOS can provide in chapter 8, giving insights on the advantages of using such technology. Also, we discussed the limitations we would face during the integration of an RTOS. Note that there are various ways to carry out this task, depending on the specific situation, we may want to have the Control Flow Integrity enforcer and the RTOS running at the same privilege level, or we may want the RTOS to run at an intermediate level while the CFI enforcer provides security for both the RTOS and the user code.

We aim to enhance the project by providing the following implementation. In our idea, the Control Flow Integrity enforcer is the only code that runs at M-mode and it is responsible for managing machine-level traps as well as forward and backward edge controls. The selected RTOS, be it *FreeRTOS*, *Zephyr RTOS*, or any other RTOS would run in supervisor mode. This is because we want the RTOS to be less privileged than the CFI enforcer but more privileged than the user code. Lastly, the user code would run in U-mode, as already described in this thesis. Figure 9.1 depicts an abstraction of the described implementations where green, orange, and red colors represent machine, supervisor, and user modes, respectively.

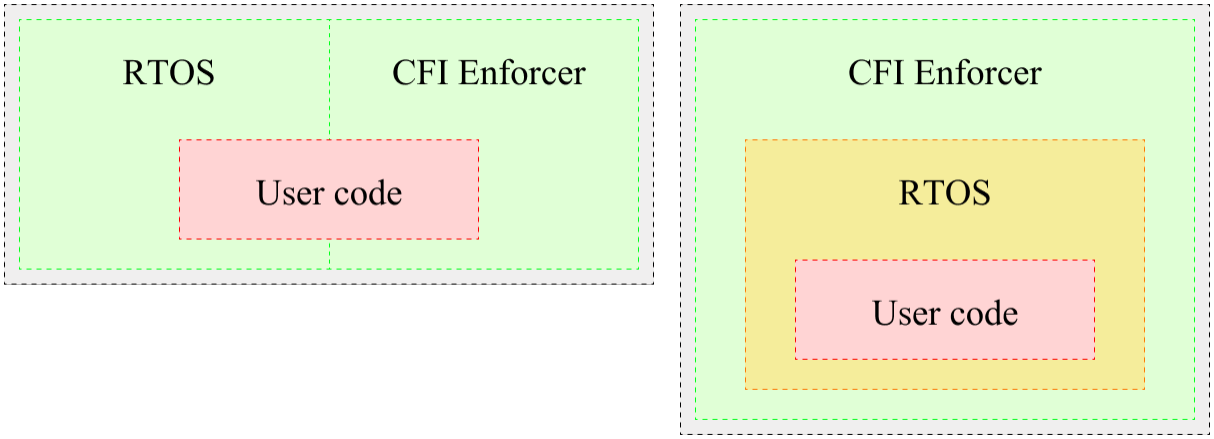


Figure 9.1: Abstraction of possible RTOS implementations

To achieve this result, the first thing we need to do is add the letter *S* to the Instruction Set Architecture we want to use, making it *RV32IMCS_ZICSR* to add supervisor capabilities to our *RISC-V* ISA.

For the second step, we need to import the files required by the RTOS. Since many RTOSes are designed with a modular approach, not every file is necessary. For example, we may want to import only the kernel itself, or if we need networking capabilities, we would need to import the connectivity-related source files and so on.

Lastly, we need to modify the instrumentation phase in two ways. Firstly, we need to add, as a target for instrumentation, the source code of the RTOS if we wish to enforce Control Flow Integrity on it. This can be done by simply adding the files related to the RTOS to the list *files_to_instrument* inside the *flasher.py* file. Secondly, we need to add the regexes and instrumentation capabilities to delegate CSR instructions to the interrupt vector table of the project. This is needed if we want the RTOS to run as supervisor since, from such a privilege level, it has no access to Control and Status Registers. A solution to this implementation is deeply discussed in section 8.4 where we list each step needed to effectively modify the instrumentation phase to address the CSR access limitation.

In conclusion, to make the building process easier, we may add a simple parameter to the input of *flasher.py*. As of now, the file can be launched with the commands described in section 5.3 but, by simply adding a parameter like *RTOS=(y/n)*, we could launch *flasher.py* with *python3 flasher.py*

command rtos. Such simple modification would make it easier to either include the RTOS files or not, thus providing a simple and fast building process for any kind of project.

10 Conclusions

This thesis discussed the design, development, and analysis of a Control Flow Integrity enforcer for *RISC-V*-based embedded devices which poses itself as a significant contribution to enhancing security in resource-constrained environments. The project successfully addresses key challenges related to embedded device security, including mitigating control flow hijacking attacks such as *Return-Oriented Programming* and *Jump-Oriented Programming*. With the integration of advanced mechanisms like a shadow stack and Control Flow Graph, the project ensures robust enforcement of Control Flow Integrity, a critical step in preventing unauthorized execution flows or arbitrary code execution.

In chapters 2 and 3, we provided a comprehensive description of control flow hijacking attacks, describing how they work and why they pose such a threat to the modern-day security of embedded systems. Moreover, we described the currently available security measures to mitigate this class of cyber threats. Lastly, we discussed the project's threat model, highlighting the scope and limitations of the provided security features.

In chapter 5, we have seen the design and implementation processes highlighting their key features. We have also seen how the project adheres to the principles of lightweight and efficient security, making it suitable for devices with limited computational resources. The system leverages the flexibility and extensibility of the *RISC-V* Instruction Set Architecture to create a tailored solution that balances security and performance. The proposed infrastructure achieves this balance through meticulous code instrumentation, optimized Physical Memory Protection configurations, and efficient trap management mechanisms. Additionally, the modular design of the project ensures that it remains adaptable to various use cases and allows for further customization. Lastly, with the Proof of Concept presented in section 5.11, we demonstrated the project's ability to effectively detect and mitigate attacks that aim at disrupting the code execution flow.

Chapter 6 was focused on providing an in-depth view of the testing methodologies used during the project's security assessment. Additionally, we proved how the Control Flow Integrity enforcer can effectively protect the code from control flow hijacking attacks. Also, we discussed the project's weak points, describing its limitations in protecting from specific attacks and discussing potential enhancements to provide a higher degree of security.

A comprehensive performance analysis presented in chapter 7 illustrates the practical viability of incorporating the project into real-world applications while maintaining high levels of system efficiency. The analysis delves into the average overhead associated with the implementation, highlighting the importance of this metric in evaluating its effectiveness. We explored various optimization techniques specifically designed to mitigate elevated overheads that may arise in certain unusual scenarios. By addressing these potential challenges, we aim to ensure that the proposed solution can be seamlessly integrated into existing systems, ultimately enhancing overall performance while minimizing any adverse effects.

The purpose of chapter 8 was to provide an implementation idea to integrate the system with widely used Real-Time Operating Systems such as *FreeRTOS* and *Zephyr RTOS*. Such integration could enhance the project's versatility and readiness for broader adoption, allowing for the execution of more complex tasks. We focused on this implementation because many real-world applications are highly dependent on the features provided by RTOSes to carry out their tasks.

Lastly, the thesis also identifies several areas for future exploration, including the incorporation of additional security mechanisms, further code optimization, and exhaustive testing under diverse operational conditions in chapter 9. These potential enhancements aim to refine the system and broaden its applicability, extending its impact on the security of embedded systems in the rapidly expanding Internet of Things landscape.

In conclusion, the project discussed in this thesis marks a significant milestone in the realm of embedded system security. This innovative approach introduces practical solutions specifically de-

signed to protect *RISC-V* microcontrollers from a variety of emerging cyber threats that have been increasingly targeting these systems. By tackling not just the immediate security challenges of these microcontrollers, but also establishing a foundation for future progress, it embodies a proactive approach to cybersecurity.

A key achievement is the focus on safeguarding untrusted code from control flow hijacking attacks, a common and dangerous threat in the embedded systems landscape. The developed infrastructure has been meticulously designed to ensure that security measures do not detrimentally affect the performance of resource-constrained systems. This balance between security and efficiency is critical, as many embedded devices operate under stringent resource limitations.

Furthermore, this work highlights the increasing necessity of secure and efficient design principles within embedded device infrastructures. As embedded systems continue to integrate deeper into various aspects of modern technology ecosystems, ranging from consumer electronics to critical infrastructure, the importance of robust security mechanisms cannot be overstated. The findings and advancements brought forth by this project not only enhance the immediate security posture of *RISC-V* microcontrollers but also serve as a foundational framework for ongoing innovation in the field, ultimately contributing to a safer and more secure technological landscape.

Bibliography

[1] Davide Moletta. Project's GitHub repository. https://github.com/davide-moletta/RISC-V-TE.

[2] De Asmit, Basu Aditya, Ghosh Swaroop and Jaeger Trent. FIXER: Flow integrity extensions for embedded RISC-V. *IEEE*, pages 348–353, 2019.

[3] Degani Luca, Salehi Majid, Martinelli Fabio and Crispo Bruno. μ IPS: Software-Based Intrusion Prevention for Bare-Metal Embedded Systems. *Springer*, pages 311–331, 2023.

[4] Espressif. ESP32-C3-DevKitM-1. https://docs.espressif.com/projects/esp-idf/en/v5.2/esp32c3/hw-reference/esp32c3/user-guide-devkitm-1.html. Accessed 21/10/2024.

[5] Espressif. IDF FreeRTOS. https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/freertos_idf.html. Accessed 09/11/2024.

[6] Fabian Thomas, Lorenz Hetterich, Ruiyi Zhang, Daniel Weber, Lukas Gerlach and Michael Schwarz. RISCVuzz: Discovering Architectural CPU Vulnerabilities via Differential Hardware Fuzzing. *CISPA Helmholtz Center for Information Security*, 2024.

[7] Free Software Foundation. GNU General Public License (GPL) v3.0. https://www.gnu.org/licenses/gpl-3.0.html. Accessed 13/11/2024.

[8] Fu Anmin, Ding Weijia, Kuang Boyu, Li Qianmu, Susilo Willy and Zhang Yuqing. FH-CFI: Fine-grained hardware-assisted control flow integrity for ARM-based IoT devices. *Elsevier*, 2022.

[9] Gisu Yeo, Yeryeong Kim, Suhyeon Song and Donghyun Kwon. Efficient CFI Enforcement for Embedded Systems Using ARM TrustZone-M. *IEEE*, 2022.

[10] High integrity systems. SAFERTOS. https://www.highintegritysystems.com/safertos/. Accessed 09/11/2024.

[11] Jaloyan Georges-Axel, Markantonakis Konstantinos, Akram Raja Naeem, Robin David, Mayes Keith and Naccache David. Return-Oriented Programming on RISC-V. *ACM*, 2020.

[12] Matthieu Baty, Guilaume Hiet and Pierre Wilke. Work in progress: A formally verified shadow stack for RISC-V. *Institute for Research in Computer Science and Random Systems IRISA*, 2022.

[13] MIT CSAIL. Kôika: A Core Language for Rule-Based Hardware Design. https://github.com/mit-plv/koika. Accessed 21/11/2024.

[14] Moghadam Vahid Eftekhari, Prinetto Paolo and Roascio Gianluca. Real-Time Control-Flow Integrity for Multicore Mixed-Criticality IoT Systems. *IEEE*, pages 1–4, 2022.

[15] Real Time Engineers. FreeRTOS: Real-time operating system for microcontrollers and small microprocessors. https://www.freertos.org/. Accessed 08/11/2024.

[16] RISC-V International. RISC-V Instruction Set Architecture. https://riscv.org/about/. Accessed 20/10/2024.

[17] RISC-V International. RISC-V Unprivileged and Privileged Specifications. https://riscv.org/technical/specifications/. Accessed 20/10/2024.

[18] Sergey Lyubka. Espressif ESP32 flashing utility. https://github.com/cpq/esputil. Accessed 13/11/2024.

[19] Sergey Lyubka. mdk: A baremetal, single header ESP32/ESP32C3 SDK. https://github.com/cpq/mdk. Accessed 21/10/2024.

[20] Valentina Forte, Nicolò Maunero, Paolo Prinetto and Gianluca Roascio. PROLEPSIS: Binary Analysis and Instrumentation of IoT Software for Control-Flow Integrity. *International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*, 2021.

[21] Werner Mario, Unterluggauer Thomas, Schaffenrath David and Mangard Stefan. Sponge-based control-flow protection for IoT devices. *IEEE*, pages 214–226, 2018.

[22] Wind River Systems. Zephyr Project: a scalable real-time operating system. https://zephyrproject.org/. Accessed 27/11/2024.

[23] xpack project. The xPack GNU RISC-V Embedded GCC. https://xpack-dev-tools.github.io/riscv-none-elf-gcc-xpack/. Accessed 03/11/2024.

[24] Zhang Jiliang, Qi Binhang, Qin Zheng and Qu Gang. HCIC: Hardware-assisted control-flow integrity checking. *IEEE Internet of Things Journal*, 6:458–471, 2018.