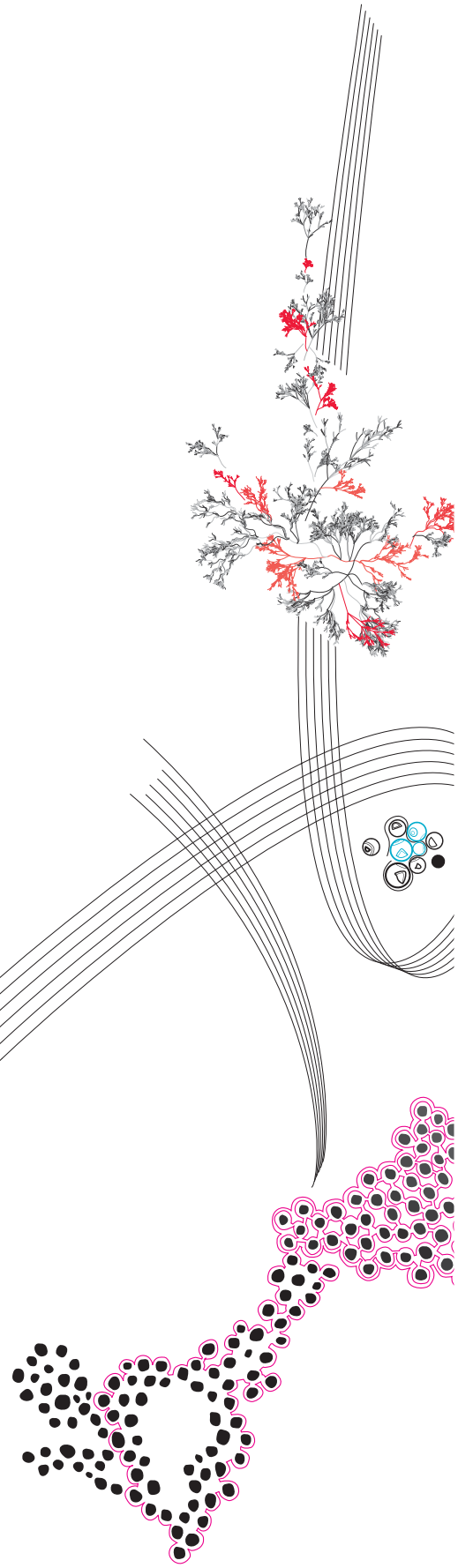


MSc Embedded Systems
Final Project

Fault Injection Attacks on Trusted Execution for RISC-V Cores

Remco van Dijk

Chair: Dr. Ir. Marco Ottavi
Supervisor: Bruno Endres Forlin
External Member: Dr. Ir. Andrea Continella

December, 2024

Computer Architecture for Embedded Systems Group
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Acknowledgments

This work would not have been possible without all of the people who have contributed their support over the course of this academic endeavor, be it professionally or personally.

First of all, I would like to express my gratitude to Dr. ir. Marco Ottavi, who introduced me to this field of study by teaching his course 'Dependable Computing Systems', which I would recommend to anyone who is even slightly interested in this subject. Furthermore, I would like to thank Marco for offering the study of fault injection attacks as a thesis; it is a very unique and interesting subject, which has been a joy to dive into and even opened up exciting new career opportunities.

I am especially indebted to Bruno Endres Forlin, whose dedication, support and expertise have helped me tremendously to expand both my technical and academic knowledge. Bruno, even though you are a guy who likes to keep busy, you were (almost) never too busy to have a discussion, to help me understand a new subject or to give me advice in any way. We have spent many hours together over the course of this thesis and for that I am extremely thankful, I do not think anyone could ask for a more involved supervisor.

I would also like to thank Dr. ir. Andrea Continella for being the external member to this graduation committee. Your feedback from a systems security perspective was invaluable and helped to make this work accessible to a broader audience. To fully address the threat posed by fault injection attacks, all engineers along the hardware-software pipeline need to be aware of its consequences and countermeasures.

Additionally, I would like to thank all of the people at the CAES group, who treated me as their colleague, despite just being here as a Master student. I especially want to thank my fellow Master and Bachelor students at the group, with whom I have shared an office for the past year or so. I feel that we made each other's work more bearable by sharing some lighthearted moments and by offering each other a second pair of eyes when asked.

Finally, I am grateful to my family and friends, who are a great support to me in a personal (and sometimes professional) way. You have all contributed greatly to my morale, without which it would not have been possible to achieve this milestone in my professional life.

Contents

Abstract

In recent years, the open RISC-V Instruction Set Architecture (ISA) has seen increasing adoption in the industry. However, as RISC-V grows in popularity, it also becomes a greater target for malicious behavior. Therefore, there is a need to facilitate the secure execution of software, especially in embedded systems, where RISC-V is gaining the most popularity. This is commonly achieved by making use of a Trusted Execution Environment (TEE). However, embedded devices often operate 'in the field', where it is possible for an attacker to gain physical access to the device in question, which enables physical attacks involving Fault Injection (FI) and Side-Channel Analysis (SCA).

This research performs a deep dive into the world of RISC-V cores with respect to their vulnerability to FI attacks using a commonly available FI method called 'clock glitching', where the attacker takes control over the clock signal and modifies it to cause a timing violation, resulting in faults such as skipped instructions. In this work, it has been confirmed that TEEs on RISC-V are generally vulnerable to FI attacks targeting Control and Status Register (CSR) access instructions, since TEEs are commonly reliant on RISC-V's Physical Memory Protection (PMP) extension to provide isolation in memory, which needs to be configured through CSRs, as specified by the RISC-V ISA.

However, in contrast to previous works that generally focus their efforts on a single core or device, this work shows that susceptibility to FI attacks is highly dependent on microarchitectural differences. Performing the same attack on different RISC-V cores using nearly identical firmware leads to vastly different results, meaning that a TEE is at most as secure as the hardware that it is running on when a system's environment enables physical attacks, which is often the case in embedded systems.

Keywords: RISC-V, Ibex, Fault Injection, Clock Glitch, Trusted Execution Environment

Chapter 1

Introduction

As of late, the RISC-V Instruction Set Architecture (ISA) is seeing a significant increase in popularity due to its open-source nature. Especially in the realm of embedded systems, RISC-V cores are widely being adopted. However, when a technology sees an increase in adoption, it generally receives an increasing amount of unwanted attention from attackers. Since computer resources are usually limited in an embedded system, but security is no less of a concern, the concept of a Trusted Execution Environment (TEE) is introduced. The TEE can be seen as a lightweight operating system that is able to support the concurrent execution of multiple user-level applications, while providing a certain guarantee of separation, such that these (untrusted) user applications are not able to tamper with other parts of the system. An example of its use can be found when looking at facilities for over-the-air firmware updates [9], where TEEs are commonly employed. Additionally, embedded systems are often deployed in uncontrolled environments where a potential attacker is able to gain physical access to the device, leading to the possibility of physical attacks involving Fault Injection (FI) and Side-Channel Analysis (SCA).

As a recent example of the implications that such attacks present, a team of security researchers performed an FI attack on the System-on-Chip (SoC) used in Tesla's infotainment system, which could be used to unlock certain locked features, such as full self driving mode [47]. A less prominent example of a real-life FI attack is the one described by Lu [29], where the cryptographic processor of a PlayStation Vita is compromised using a voltage glitch to gain arbitrary code execution at boot-time, enabling the attacker to dump the ROM contents. These examples of FI attacks are not specifically targeting RISC-V, but nevertheless illustrate the significance of FI attacks on embedded systems.

Continuing on the theme of attacks that target other ISAs, a trend starts to form: multiple sophisticated attacks on ARM and x86 have been demonstrated in the past [47, 35, 11], which are often capable of bypassing a TEE or secure boot mechanism despite having to consider a number of complicating circumstances, such as using a well-established industry standard TEE (ARM TrustZone) or dealing with closed-source firm- and hardware. In contrast, a recent RISC-V related study by Nashimoto et al. [31] shows how a Proof-of-Concept (PoC) TEE can be bypassed using FI by targeting instructions that control the underlying Physical Memory Protection (PMP) mechanism. In addition to the PoC-TEE, it is theorized how a more sophisticated TEE, Keystone [23], could be bypassed. Their theory is based on the fact that both their PoC-TEE and Keystone make use of the PMP unit to provide separation in memory, which continually has to be configured correctly by writing to so-called Control and Status Registers (CSR), which relies on the same set of CSR-specific instructions. In essence, this would mean if one is able to successfully attack a simple PoC-TEE, then it must be possible to attack a more sophisticated TEE such as

Keystone. However, one of the key assumptions is that both TEEs are running on the same core microarchitecture; it is yet unknown what the implications would be of using the same attack scheme to target different microarchitectures.

From the current state of research, it becomes clear that particularly in the case of RISC-V, the issue is not that it is inherently more or less vulnerable to physical attacks than other ISAs, such as ARM or x86, but far less is known about the security implications of these attacks on cores that implement the RISC-V ISA. Additionally, previous works generally do not consider the impact of microarchitectural differences when assessing vulnerability to FI attacks, especially when it comes to RISC-V.

Therefore, the following research question is posed:

How can a TEE be compromised using fault injection on embedded RISC-V cores and how is this affected by different microarchitectures?

To support the main research question, the following subquestions are formulated:

1. *What are effective attack vectors for a TEE on RISC-V?*
2. *How are the effects of fault injections influenced by different microarchitectures?*
3. *Which strategies could be employed to reduce the effectiveness of these attacks?*

Considering the state of the art, it is hypothesized that a TEE - without considering any direct software vulnerabilities - on an embedded RISC-V core can be compromised using an FI attack by directly or indirectly targeting underlying hardware-defined security mechanisms, such as PMP. However, the susceptibility to FI attacks will likely depend heavily on the type of hardware that is used, as FI techniques rely on physical phenomena to cause hardware faults. Since architectures are often very differently implemented and laid out in silicon, it is expected that different architectures will show different behavior when exposed to an FI technique.

In summary, compared to well established ISAs, such as x86 or ARM, there are still many unknowns about the security implications of physical attacks on RISC-V and its different hardware implementations. This research attempts to contribute to the state of the art by identifying different attack vectors for FI attacks on TEEs for RISC-V, investigating the influence of different microarchitectures on attacks utilizing those attack vectors and proposing specific countermeasures.

Chapter 2

Background

2.1 Fault Injection

Fault Injection (FI) attacks were first demonstrated in 1997 by Boneh et al. [6] as a way to compromise cryptographic hardware. Since then, a multitude of methods to purposefully induce faults in integrated circuits has been introduced [2]. Nowadays, such methods are not only used to attack cryptographic accelerators [3], but also for circumventing secure boot mechanisms [11, 25], TEEs [31] and more types of secure software systems, as FI attacks can offer a comparatively simple backdoor into an otherwise secure system if the hardware is not properly secured.

2.1.1 Overview

Over the years, many techniques have been explored for injecting faults into integrated circuits, either to compromise their security or to analyse their dependability in electromagnetically harsh environments such as space. Below is an overview of methods that are commonly used today.

Voltage Glitching: Integrated circuits are often required to operate under certain conditions, if these conditions are not met, then their correct operation is not guaranteed. Violating the required operating conditions is often an easy target for attackers looking to inject faults in a circuit. With voltage glitching, the idea is to manipulate the power supply of a chip such that it either receives very short high voltage spikes or a voltage that is too low. Short high voltage spikes have been shown to cause undefined behavior in latches of flip-flops [21], while undervolting usually results in a continuous effect where multiple faults occur. Voltage glitching attacks are commonly applicable in the embedded space, where a physical attacker has close access to the power supply of the chip. Some great benefits of voltage glitching are its ease of use and the low cost to mount an attack, since it usually only requires modifications to the power supply using inexpensive parts. However, it does not offer much precision, since power supply disturbances will generally affect the whole chip.

Clock Glitching: Synchronous digital circuits receive either an internal or external clock signal that is used to synchronize its operation. A typical clock glitching attack relies on the manipulation of an external clock signal to violate timing requirements and cause undefined behavior. To achieve this, the attacker usually generates a malicious clock signal with a signal generator of their choice and feeds this into the target chip. The benefits of this method are similar to the ones of voltage glitching, albeit with a slightly higher cost

due to the need for hardware that is able to generate a glitched clock signal with a sufficient degree of configurability; a Field-Programmable Gate Array (FPGA) is a commonly used component in such a setup. It also offers slightly more precision, since different parts of a chip could make use of different clock domains, allowing the attacker to choose which clock domain to target.

Optical Fault Injection (OFI): The effects on transistors induced by ionizing radiation have been known for a long time [15, 5] and are a great risk to the dependability of exposed chips. This knowledge led to the first documented attacks on a CMOS chip using either a simple laser pointer or a flashgun in combination with a lens [41]. OFI is one of the first fault injection methods to be explored because of its higher precision and relatively low equipment cost. An attacker only needs an exposed die, a visible light source, such as a camera flash, and a lens to precisely direct the flash at the target. The main downside of this method is that it can be cumbersome to prepare the target device for fault injection, requiring at least decapsulation and possibly delayering of the silicon as the visible light is not always able to penetrate the silicon to a sufficient depth. Since delayering is not always possible without damaging the circuitry, OFI is not always feasible.

Laser Fault Injection (LFI): LFI is a rather similar method to OFI in that it is able to direct an electromagnetic wave at a precise physical location in a circuit, but it makes use of an infrared laser instead of a visible light source. It seems to be that LFI is a significantly more popular method compared to OFI, since there are a number of commercial products available to perform it. However, these setups are comparatively quite expensive (up to $\sim$ 100.000 USD) [7] and typically consist of specialized equipment, such as a laser source, a lens, a positioning table and a controller. It should be noted that, although less is known about them, there have been a number of recent efforts to significantly reduce the cost and make LFI more openly accessible [13, 19], potentially bringing down cost of a setup to hundreds of USD instead of hundreds of thousands.

X-ray Fault Injection (XFI): XFI is again a quite similar method to OFI (and LFI) for the same reason that LFI is similar to OFI; where LFI makes use of a laser beam, XFI uses a nanofocused X-ray beam to target components on a nanometer scale. Besides its precision, another great advantage of this method is that the package of a typical chip is invisible to X-ray waves, meaning decapsulation is no longer required. However, it is arguably the most expensive documented method of fault injection, ranging in the millions of USD [7], rendering it highly impractical for most attacks. However, even though the cost and specialized equipment needed might make this method seem highly impractical, Nasr-Eddine et al. [42] have recently shown that such attacks are feasible and could be considered a threat, especially in situations where the possible gain of an attacker outweighs the initial cost of an XFI setup.

Electromagnetic Fault Injection (EMFI): EMFI relies on the effects that EM emanations have on both analog and digital logic. In the case of digital logic specifically, the goal is to inject faults by inducing a short but intense transient current in the circuit, such that operations are affected during a single clock cycle [37]. These transient currents are often induced using a high voltage pulse generator in combination with an injection probe, which are relatively inexpensive components; low-cost pulse generators can be obtained for around 3.300 USD and injection probes can be made from simple electrical components [7]. Another benefit of this method is that decapsulation is usually not required, since common packaging materials are invisible to EM emanations. To add an additional degree of precision, the target chip may be mounted on a motorized positioning table. A downside of EMFI is that the most expensive part, the pulse generator, can become more expensive when a higher precision unit is required, the price of which ranges between 10.000-20.000

USD [7].

Body Biasing Injection (BBI): BBI is an FI method that generally relies on injecting a high voltage pulse on the backside of a chip that introduces a static bias within the die for a short period of time [30, 33]. Similarly to EMFI, the most expensive part of the setup for this method is an external pulse generator and power supply; O'Flynn [33] shows how a viable setup can be constructed for around 750 USD. However, when a higher precision pulse is needed, the cost becomes the same as high-precision EMFI setups, since similar pulse generators are used. In contrast with EMFI, decapsulation may be required depending on the type of chip packaging that is used. Most research of BBI indicates that decapsulation is necessary, but O'Flynn [33] has recently shown that it is not needed on Wafer-Level Chip-Scale Packaging, where the backside of the die is inherently exposed. Finally, similarly to EMFI, an additional degree of precision can be obtained by targeting different physical locations on the chip using a motorized positioning table.

2.1.2 Challenges

When performing fault injection attacks, there are numerous challenges to be addressed by the attacker depending on what type of fault injection technique they have chosen. Some techniques may involve more specialized (and expensive) equipment, while others only require cheap, readily available parts.

Ease of Use: To even start performing fault injections on a circuit, an attacker often needs to make various physical modifications to either the chip itself, the package or the environment. The amount of effort that has to be expended before fault injections can be attempted can differ greatly between FI techniques. One technique may only require the attacker to attach wires to certain pins, while another may require much more invasive practices, such as mechanical or chemical decapsulation to expose the die's surface or deeper layers.

Cost Efficiency: Different FI methods require different setups, some of which could be made up of some wires and other common lab supplies, while others may require specialized and expensive equipment, such as precise lasers. Therefore, when assessing the vulnerability of a system against FI attacks, the considered techniques should be limited to the ones that have a cost which is proportional to the value of the target to a potential attacker.

Spatial Precision: Different FI techniques have different physical effects on a circuit. Some of them might be rather imprecise and affect the entire chip area, while others might be able to target just a single transistor. It is said that a higher spatial precision generally results in better control over the injected fault.

Temporal Precision: A fault injection most often requires the generation of some anomalous signal or pulse. For example, a glitched clock signal or a pulse that is sent to an EM probe or body biasing probe. A better capability to control the shape of that signal, such as being able to configure it within a higher range of frequencies, often translates into a greater level of control (temporal precision) and being able to target logic that operates at a higher frequency

Table 2.1 shows each of the challenges mentioned above as a metric by which to assess the desirability of a certain FI method. They are quantified on a scale from 1 to 5, where a higher score is generally more desirable. In figure 2.1 each of the FI methods described before is scored on each metric according to table 2.1. When choosing which method to use, one can fill in their needs for each metric and use this figure to determine which method is most suitable.

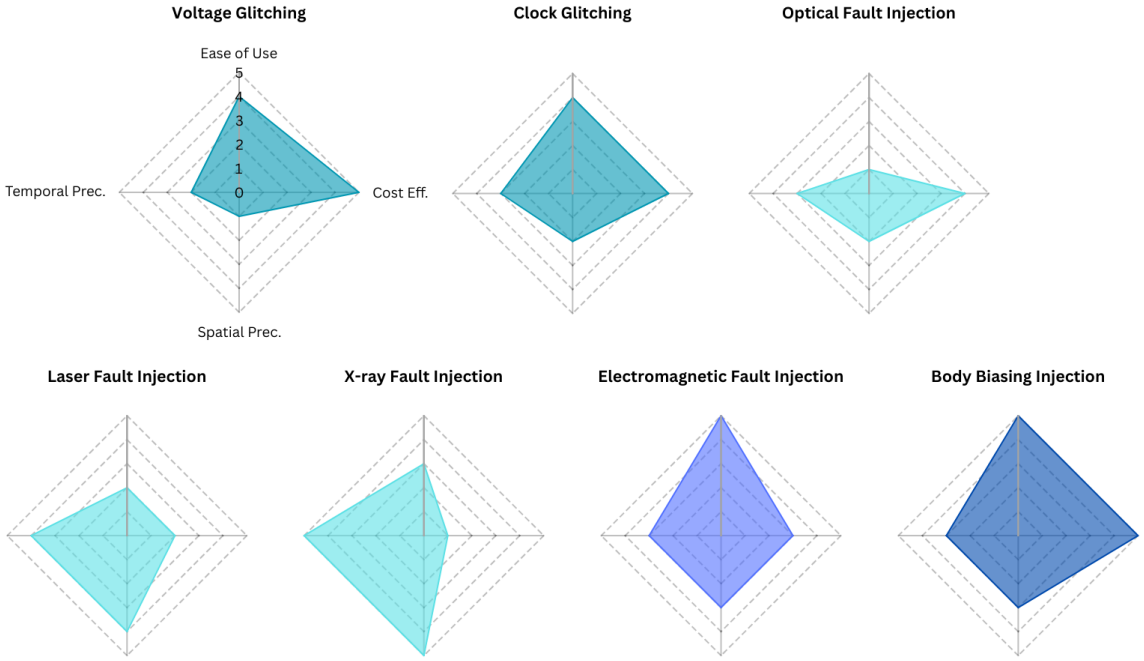


FIGURE 2.1: Comparison of FI Methods Based on Metrics

Metric \ Score	1	2	3	4	5
Ease of Use	At least decapsulation and possibly delayering are needed	At least decapsulation is needed	Knowledge of chip layout is needed	Modification of on-board components such as clock or voltage source is needed	No or light modifications are needed such as attaching wires or probes
Cost Efficiency	cost > \$100k	\$100k < cost < \$10k	\$10k < cost < \$1k	\$1k < cost < \$100	\$100 < cost < \$0
Spatial Precision	Affects the entire chip with little or no possibility of adjustment	Different clock domains can be targeted	Can be positioned over a general area of the target chip	Can target specific chip areas at a millimeter or micrometer scale	Can target components on a nanometer scale
Temporal Precision	Signal/pulse shape cannot be adjusted	Signal/pulse shape could be obstructed or filtered by on-board components	Effectiveness is fully dependent on the pulse generator used	Signal/pulse shape can be adjusted on a nanosecond scale	Signal/pulse shape can be adjusted arbitrarily or is irrelevant

TABLE 2.1: Scoring of each Challenge Metric for FI Methods

2.1.3 Tools

Currently, one of the most popular platforms for fault injection is the ChipWhisperer ecosystem offered by NewAE Technology Inc [17]. Their family of relatively low cost devices covers a large subset of the commonly known FI methods described in section 2.1.1, namely voltage glitching, clock glitching, EMFI and BBI. The low cost nature of their products makes it approachable for hardware designers and researchers to evaluate their systems against FI attacks; consequently, this is the platform of choice in this research.

When considering other FI methods, such as OFI, LFI or even XFI, other tool platforms should be considered, as these methods are more specialised and often require significantly more expensive hardware. For example, Petryk et al. [34] show that the overwhelming majority of research into OFI and LFI is performed using tools offered by Riscure. Even though these tools are popular and well-documented, they are quite expensive. Therefore, it should be noted that there have been recent efforts to offer similar tools at cheaper price [19, 13]. Although these tools are not as mature, they could offer a decent alternative for researchers and developers who are working with a lower budget.

2.2 Trusted Execution Environments

A Trusted Execution Environment (TEE) [36] is defined as a tamper-resistant software, often referred to as the monitor, that facilitates the isolation and concurrent execution of untrusted software, often referred to as enclaves. Examples of enclaves include Operating Systems (OS), facilities for over-the-air (OTA) firmware updates [9] and any application-level (or user-level) software.

In a TEE, isolation is achieved by separating enclaves from each other, following the principles of a so-called separation kernel, such that they are unable to tamper with other enclaves or the monitor itself. As defined by the Separation Kernel Protection Profile [14] and summarized by [50] and [36], a TEE is thereby required to comply with four main security policies:

1. **Spatial separation:** Physical memory assigned to one enclave cannot be read or written to by any other enclave.

2. **Temporal separation:** Shared resources, such as timers, cannot be used by a malicious enclave to gather information about other enclaves or the monitor.

3. **Inter-enclave communication:** Communication between enclaves, for example through shared memory or using certain registers, is not possible unless explicit permission is given by the monitor.

4. **Fault isolation:** A fault in one enclave cannot propagate to other enclaves, meaning that a security breach in one enclave cannot affect the other enclaves.

To enforce these requirements, the hardware being used to run a TEE must provide appropriate primitives. For example, a memory protection unit and privilege levels. Generally, these primitives are defined differently for each ISA that supports trusted execution. However, it could even be that the primitives differ per hardware implementation of an ISA. For example, Intel and AMD are both companies that manufacture cores implementing the x86 ISA, but both use different specifications for trusted execution.

When these primitives do not exist in a system or if they are not used by the developer, it is generally the case that applications are able to freely access all physical memory on the system. This may not be a problem in simple systems that just execute a single task, but such a system cannot be considered secure.

2.2.1 ARM TrustZone

To draw a proper comparison between TEEs on more established ISAs and those on RISC-V, ARM and its TrustZone specification are chosen. TrustZone [32] is chosen for both its wide adoption in industry, as well as its similarity to RISC-V based TEEs. TrustZone's M-profile is especially similar, as it is aimed towards secure microcontrollers that are often used in secure embedded systems. Other system-wide security specifications include Intel's Software Guard Extensions and Trusted Execution Technology [8] and AMD Secure Encrypted Virtualization [1], all of which are specifications for x86 systems, which tend to be targeted less towards embedded systems and therefore less suitable for comparison against RISC-V based TEEs.

TrustZone is a system-wide specification that is used to architect a hardware-enforced TEE. It defines that the system is divided into two worlds: the secure world, where security-critical tasks such as cryptographic algorithms are executed, and the normal world, which

is used for general user applications. The goal of this separation is to guarantee a strong isolation between 'secure' and 'normal' operations [24].

The main separation, which is specifically important in the context of TEEs, is separation in memory, which is achieved by the Memory Management Unit (MMU); in TrustZone systems, both worlds have their own MMU. Each MMU is responsible for providing memory protection and virtual memory management, which is enforced during the translation of virtual memory addresses to their physical counterparts. To guarantee this separation, each world (or MMU) must maintain its own page tables and memory protection configuration.

2.2.2 TEEs for RISC-V

The TEE ecosystem for RISC-V is notably less developed than the ones for ISAs such as ARM and x86, which have been on the market for a significantly longer time. Even though the specification for hardware-defined mechanisms related to TEEs is well-defined for RISC-V, software libraries are still largely fragmented. However, the projects that are currently in development do show a number of similarities that are unavoidable when building a TEE for generic RISC-V cores. These similarities are analyzed to construct an attack scheme that will not just apply to a specific TEE implementation, but to the shared elements between RISC-V TEEs.

One of the more prominent projects for architecting TEEs on RISC-V is Keystone [23], which is fully open source and targets more powerful application class RISC-V processors, as it supports running a secure TEE monitor, a rich OS such as Linux, as well as user applications, each in their separate privilege mode. Since embedded RISC-V cores usually support at most 2 privilege modes, they are considered to be incompatible with Keystone. However, there do exist TEE-related projects that are compatible with embedded RISC-V cores, such as HexFive's MultiZone API [38] and its open-source implementation: OpenMZ [18]. The MultiZone API aims to define TrustZone-like primitives to provide a compatibility layer between existing TrustZone-based designs and RISC-V hardware. Another recent development is Penglai Enclave [12], which is an open source TEE framework for RISC-V, it can be adapted to both application class and embedded processors.

Even though there are many competing projects for TEEs on RISC-V, there are a number of similarities between them, which means targeting those similarities in an attack enables it to affect most, if not all, RISC-V TEEs. In the context of an attack, the most important similarities include the use of RISC-V's PMP extension and at minimum 2 privilege modes, where a 'secure' TEE monitor is running in a high privilege mode and 'non-secure' user-level applications are running in a lower privilege mode.

Figure 2.2 shows the stack of different components required to construct a minimal TEE on RISC-V, increasing in privilege level from the bottom upwards. It starts with a collection of trusted hardware, consisting of at least one or more RISC-V cores that implement the PMP extension. Other (custom) hardware extensions may also be present to enhance the system's security, such as a cryptographic accelerator. Below the hardware is the TEE monitor software, which runs in the highest privilege mode, often referred to as privileged execution mode, machine mode or M-mode. The monitor is responsible for the boot process and performing context switches between user apps (i.e. switching between their execution), meanwhile maintaining a correct configuration of the PMP hardware, such that user apps - which are considered to be untrusted - only have access to their assigned memory region. If one of the user apps attempts to read memory outside of its assigned region, the PMP unit will detect this and notify the monitor by means of an interrupt, allowing it to take whatever action is needed; for example terminating the app that caused

the interrupt. Finally, there are a number of user-installed software applications who are considered to be untrusted and should therefore be restricted and protected by the TEE. The apps may request action from the monitor, similarly to how one would use system calls in a Linux-like OS, by executing the `ecall` instruction; after the monitor handles the request, it uses the `mret` instruction to lower the privilege level and return to the execution of a user app.

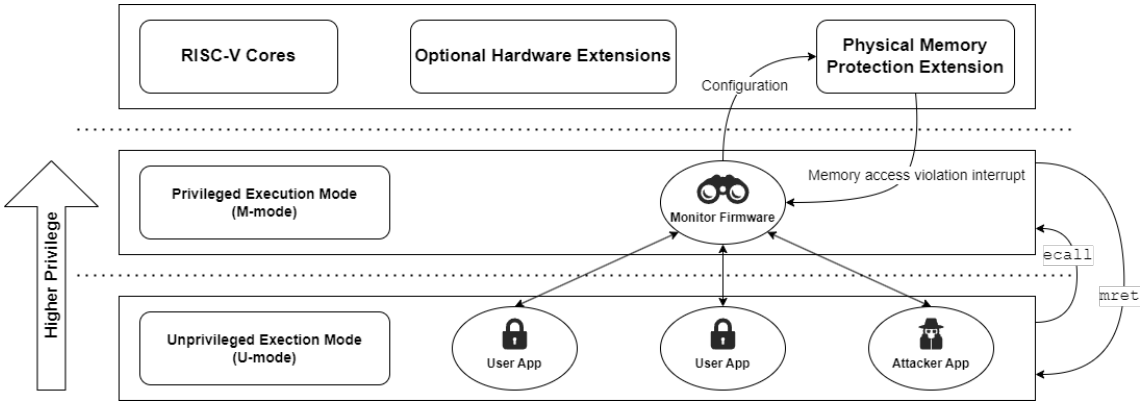


FIGURE 2.2: Hardware and Software Stack for a TEE on RISC-V

Privilege Modes: Secure processors often employ so-called privilege modes that give the firmware developer the possibility of dividing code into 2 or more levels of privilege. This enables the separation of trusted code, such as the TEE monitor, from non-trusted code, such as user applications running in a TEE.

This research deals with embedded RISC-V cores, which commonly employ 2 privilege modes: a high-privilege mode called machine mode (or M-mode) and a low-privilege mode called user mode (or U-mode). However, larger cores may employ a third privilege mode in between M-mode and U-mode called supervisor mode (or S-mode), which is usually reserved for an operating system such as Linux. Code running in a lower privilege mode is able to request certain actions from the privilege level above it by triggering an interrupt using the `ecall` instruction, after which the higher privilege mode is able to return to the lower privilege mode using the `mret` (or `sret`) instruction. On a high level, this mechanism is analogous to the concept of system calls in an operating system.

RISC-V Physical Memory Protection: The PMP extension was first introduced in the RISC-V privileged 1.10 standard [46] as a hardware primitive that provides M-mode software with the ability to impose memory access rules on lower privilege software. RISC-V cores that implement this feature expose certain Control and Status Registers (CSR) that allow the TEE monitor running in M-mode to configurably restrict U-mode (and S-mode) from accessing regions of physical memory.

Each memory region and its access rules are configured by an 8-bit value that dictates the rules, along with two 32-bit values that make up the first 32 bits of two 34-bit physical memory addresses that dictate the address range, giving a maximum granularity of 4 bytes to each region. Such a combination of configuration values is commonly referred to as a PMP entry; when the M-mode software writes the entry to its respective CSRs, each subsequent memory access is verified according to it and any other entries that were already present. An interrupt is raised when a disallowed memory access is attempted, which is then handled by the TEE monitor.

Chapter 3

Related Work

3.1 Existing Fault Injection Attacks

FI attacks are a major threat to embedded systems that are deployed in publicly accessible spaces, since a potential attacker can easily gain physical access to the hardware and make arbitrary changes to its environment, such as mounting an EM probe near it. However, even embedded systems in private spaces could be affected. For example, it has been shown that recent models of Tesla cars' on-board computer is vulnerable to FI attacks [47], allowing the owner to do anything from activating soft-locked heated seats, which is relatively innocent, to severely compromising traffic safety by activating a strictly forbidden full self-driving mode, which is only meant to be used for testing and not meant to be accessed by the end user.

Clearly, these attacks are a real threat to the security of everyday embedded systems. Therefore, this research will investigate the effects of FI attacks on them, specifically using the RISC-V ISA since it is becoming an increasingly popular choice in the world of embedded systems, while relatively little research on FI attacks has been done using different RISC-V cores, which will become clear from this chapter.

3.1.1 Fault Injection on ARM

Research into FI attacks on ARM is quite advanced when compared to RISC-V, which is in part explainable by the fact that RISC-V processors have been on the market for a significantly shorter period of time than their ARM counterparts, thus garnering more attention earlier.

To show some recent examples, Qiu et al. [35] have shown how the current standard for a TEE on ARM - TrustZone - can be breached by manipulating the dynamic voltage and frequency scaling present on multi-core ARM processors using just software. Additionally, Fanjas et al. [11] propose a methodology to perform an FI attack on a closed-source TEE for a smartphone-grade ARM processor.

Looking at the research on FI attacks that has been performed so far using ARM chips, it becomes quite clear that these attacks are rather well developed. This can be observed by looking at the attacker models used in recent studies, which describe complex and restrictive conditions for the attacker. For example, closed source TEEs have been successfully attacked on closed, consumer-grade hardware and there have even been so-called 'remote' FI attacks that do not even require the attacker to have physical access to the target device (for example Qiu et al. [35]).

3.1.2 Fault Injection on RISC-V

When it comes to RISC-V, research on its susceptibility to FI attacks has only recently taken off compared to other architectures such as ARM, because of its relatively short time in the market. It seems to be the case that much of the research focuses on FI simulation to verify certain fault tolerance methods, with only a few studies using FI as a technique to perform an attack.

One of the earlier examples is when Laurent et al. [22] investigated the security implications of FI attacks on a RISC-V core by means of Register-Transfer Level (RTL) simulation and showed the possibility of various attacks in 2019. In the same year, Werner et al. [48] theorize a variety of techniques to protect RISC-V cores against FI attacks. However, this is again supported by a simulation-based FI campaign and does not evaluate any real FI attack scenarios. A year later, Elmohr et al. [10] advanced the field by showing experimental results of applying Electro-Magnetic (EM) FI techniques on an embedded RISC-V processor, demonstrating that faults can be reliably injected into a hard RISC-V core when considering a real attack scenario. Finally, research by Nashimoto et al. [31] demonstrates an FI attack on their PoC TEE (2020) and is able to attack a more sophisticated TEE (2022), Keystone, when adapting the firmware to simplify the attack.

In summary, research into the consequences of FI attacks on RISC-V cores has only recently taken off and therefore has numerous limitations when compared to other ISAs. When drawing a comparison between RISC-V and ARM in this respect, it becomes clear that within the world of research into FI attacks, research that utilizes RISC-V technology is lagging behind the ones that have used ARM, owing to the often looser constraints in the attacker model of RISC-V related work. Since processors from both ISAs are used extensively in the embedded systems space, with RISC-V usage steadily increasing, it is imperative that more FI attack related research needs to focus on RISC-V and its different microarchitectures.

3.1.3 Fault Injection Timing

With many FI attack scenarios, across a variety of FI methods, there is often the same problem that arises when bringing the attack into practice: finding the appropriate injection timing. Since FI methods commonly make use of an external device that observes the target and injects the faults, it can be tricky to synchronize this device with the operations happening on the target.

In many cases, for example in Nashimoto's work [31], the attacker is assumed to be able to run at least some software in a lower privilege mode to allow it to produce a so-called 'trigger' to the outside world through GPIO pins or other output ports. It can be argued that this is still realistic, for example in a TEE where a user is allowed to install their own applications under a lower privilege mode. However, this still leaves the problem of finding the delay between receiving a trigger signal and the target executing its critical operation, as these are often far apart in time.

A good solution to this problem is triggering by template matching. This method has been used in practice by van Woudenberg et al. [44] to successfully target a secure microcontroller with OFI. A specific template matching external triggering tool was later presented by Beckers et al. [4]. Essentially, template matching relies on the collection of a side-channel trace, which is known to be captured during the execution of a critical section of code. After which, the target device is left to execute its code, while its side-channel trace is continually being recorded; when the earlier recorded template is then 'matched' in real-time to the target's trace, the fault injection is triggered.

However, since this method involves quite a few difficult technicalities and is not yet demonstrated in any of the previous work on RISC-V, it will not be the focus of this research. Like previous works on RISC-V, this work will make use of a trigger signal which is simply generated internally in the device under test.

3.2 Fault Injection Countermeasures

Countermeasures against FI attacks often overlap with techniques for improving the fault tolerance of computer systems, which is often necessary when a system is deployed in environments where hardware faults are likely to be induced by environmental factors. For example, the ionizing effects of radiation have been known to cause faults in semiconductors, in both a lab setting [15], as well as in real-world scenarios such as satellites [5]. Techniques for improving a (digital) circuit's resilience to faults are generally based on introducing redundancy to a certain degree. Said techniques are not only useful in environments such as space, where cosmic radiation may induce unintentional hardware faults, but also in (embedded) systems that are deployed in spaces where an attacker could gain physical access and intentionally inject faults.

Techniques for introducing redundancy can usually be categorized in 3 different categories, namely spatial redundancy, temporal redundancy and information redundancy. Spatial redundancy often involves duplicating certain components in a system that are mission-critical and/or sensitive to faults. For example, one might only need 1 processor in a system, but chooses to arrange 2 or more in parallel; this technique is often referred to as N-modular redundancy [20]. By doing so, errors can be detected or even corrected when the parallel units do not give the same output, given the same input. While spatial redundancy strategies can significantly reduce fault propagation rates without compromising execution time, it often incurs a high cost in terms of chip area usage and power consumption.

On the other hand, temporal redundancy is based on the repetition of a critical operation over time, which allows for the comparison of initial and subsequent outputs. This initially allows for error detection and could be used as a trigger for the system to start a self-recovery procedure. However, temporal redundancy does generally depend on the fault being transient; if the fault is permanent, the subsequent result will be equal to the initial, but both will be erroneous and the fault is therefore not detected. As a recent example, Villa et al. [45] have shown how temporal redundancy techniques, such as the so-called 'Checkpoint Recovery technique', can be used to detect and recover from transient hardware faults caused by the effects of ionizing radiation.

Finally, information redundancy relies on the addition of redundant bits of information to critical data, allowing for the detection and possibly correction of one or more bit-flips. A rather common way of achieving information redundancy is with error-correcting codes (ECC), which was already proposed by Hamming [16] in 1950. Since the advent of modern computing, ECC are often employed in memories of mission-critical systems in order to periodically detect and correct soft errors. Similarly to spatial redundancy techniques, the extra memory needed to store redundant bits and the circuits to encode and decode information often incur a penalty in terms of hardware cost.

3.2.1 Hardware Countermeasures

There are a number of ways to implement fault mitigation patterns in hardware. Since this work evaluates 2 specific RISC-V cores' susceptibility to fault injection attacks, their

most significant hardware countermeasures are discussed. The first core being evaluated is SiFive's E31 core [40], which is not specifically disclosed by SiFive to have any protection from hardware faults. However, the second core, being lowRISC's Ibex core [28], does support a number of significant fault tolerance methods in hardware. Since the Ibex is used as a soft core, meaning its hardware structure is synthesized and placed on an FPGA, it is possible to configure the hardware synthesis process such that its redundancy measures are enabled.

One of Ibex's major security features is dual-core lockstep, which can be categorized as both a spatial and a temporal redundancy technique. When dual-core lockstep is enabled, a second instance of the core is realized, also known as the shadow core. Outputs of the shadow core are continually checked against a delayed output of the main core; when a discrepancy is detected, a major alert is signaled, which can be used to trigger an interrupt. Such interrupts could then be used, for example by a TEE's monitor firmware, to attempt a recovery from the detected fault.

Additionally, Ibex makes use of ECC in various memories, which is categorized as information redundancy. For example, it uses ECC checking to detect errors on a read of the register file. When an error is detected, it is not corrected, but a major alert is signaled, similarly to when a mismatch is detected by the shadow core when dual core lockstep is enabled. Another place where the Ibex employs ECC is in the instruction cache; when an error is detected, the whole cache is invalidated to avoid the execution of potentially malicious code.

In conclusion, hardware measures for improving fault tolerance of digital circuits are plentiful and have been researched since the start of modern computing technology. Each one belonging to one or more categories of redundancy techniques and coming with their own benefits and trade-offs. Many more of these techniques exist than mentioned in this section, but these are some of the more notable ones to be found in the work related to this research.

3.2.2 Software Countermeasures

Aside from implementing countermeasures against fault injection in hardware, it is also quite possible and effective to implement countermeasures in software [43, 49]. Where hardware countermeasures often have a high cost in terms of chip area/power usage and require specialized hardware designs or reconfigurable hardware to be implemented, software countermeasures are often able to introduce a substantial improvement to fault tolerance with the addition of a small piece of software.

In an overarching review by Theissing et al. [43], it is shown that known software countermeasures are quite numerous and can be categorized into 2 categories: data/instruction redundancy and control flow verification. Data redundancy often relies on a simple duplication of critical instructions and data, which works by reducing the probability of the same fault occurring in all copies of the instruction/data; if the probability of faults occurring in each copy is independent, this makes for quite an effective strategy. Control flow verification techniques operate by storing a unique identifier of the jump target as a signature and verifying this after each jump; if the previously stored signature and the current code block's identifier do not match, a recovery procedure can be triggered. Theissing et al. conclude from their tests that a combination of both categories yielded the best fault tolerance while still incurring an acceptable execution time penalty.

Chapter 4

Methodology

To find an answer to the research questions and place them into context, the scenario of an FI attack by clock glitching on a TEE is described in this chapter. First, the attack scheme is laid out, describing the different steps that the attacker will take to mount a successful attack, including a motivation and detailed specification of the chosen attacker model. Finally, the PoC-TEE used in this scenario is described in more detail, showing why this case study is applicable to more well known TEEs for RISC-V, such as Keystone.

Answering the first part of the research question - 'What are effective attack vectors for a TEE on RISC-V?' - is done by observing similarities between previous works that deal with implementing TEEs on RISC-V (see section 3.1.2), implementing similar behavior in the PoC-TEE and showing a successful attack that exploits these features on a RISC-V core.

Then, in order to effectively answer the second part of the research question - 'How are the effects of fault injections influenced by different microarchitectures?' - the proposed attack scheme will be tested in two different experiments involving different RISC-V cores: the first experiment covers SiFive's E31 core and the second uses lowRISC's Ibex core. This approach will give an idea about the differences and similarities between the two cores when it comes to their vulnerability to clock glitching attacks. Since the two chosen cores are quite different, doing this will not give a definite answer as to which microarchitectural features correspond to an increase or decrease in vulnerability to hardware faults, but it should provide some interesting clues and discussion points. However, it will definitely show whether there is a significant difference worthy of further investigation between different RISC-V microarchitectures.

Finally, to answer the third part of the research question - 'Which strategies could be employed to reduce the effectiveness of these attacks?' - different methods can be argued on the basis of microarchitectural implementation details that are common between RISC-V cores capable of running TEEs.

4.1 Attack Scheme

Figure 4.1 shows a general overview of the proposed attack scheme to be considered in this research. It includes a two-part profiling phase where a copy of the target device is used with firmware modified by the attacker in order to gather information to aid in the attack, after which there is an attack phase where the previously acquired information is used to mount an attack. More detailed explanations of the profiling and attack phases can be found below in section 4.1.2 and 4.1.3 respectively.

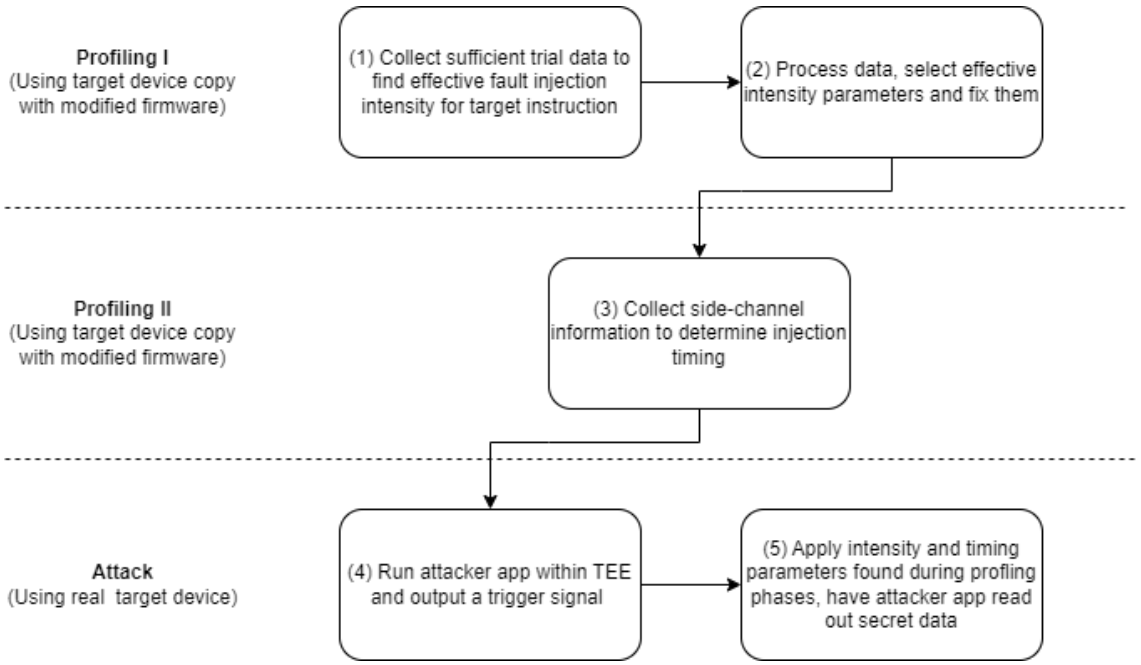


FIGURE 4.1: Flowchart of Attack Scheme

4.1.1 Attacker Model

Based on the definition of a TEE, as described in section 2.2, a TEE should guarantee spatial separation, or separation of physical memory. Therefore, it is assumed that the goal of the attacker is to access memory assigned to other user applications running in the same TEE, for example to find a key used for encryption of sensitive data by the victim application. Apart from guaranteeing isolation in various aspects, a TEE does not guarantee much else, such as confidentiality, integrity and/or authenticity of user applications running within it. Therefore, attack vectors such as tampering with the TEE firmware are not considered.

Seeing the current state of the art with respect to FI attacks on RISC-V (please refer to section 3.1.2), a rather powerful attacker model is chosen. In this scenario, the attacker is said to have the following privileges:

- The target device is physically accessible to the attacker, opening up the possibility for physical attacks.
- The target device is a common off-the-shelf component, in which case the attacker is able to obtain one or more duplicates of the device to use for testing.
- The TEE's monitor firmware is openly accessible (open source).
- The attacker is able to install or execute their own firmware application in the TEE with U-mode privileges.
- The attacker application can make requests to the TEE's monitor that eventually cause the victim application to run.
- The attacker application has access to I/O facilities such as GPIO pins and serial (UART) I/O. In practice the TEE could block U-mode applications from accessing memory mapped I/O facilities using the PMP unit, but with this attacker model it is allowed.

4.1.2 Profiling Phase

During the profiling phase, a modified version of the software is uploaded to the target such that information can be gathered by performing fault injections on it. Information is gathered about effective fault intensity and timing, which is later used during the attack phase.

The first part of the profiling phase, as shown in figure 4.1, deals with finding effective intensity parameters for fault injection on the selected target instruction. The attacker does this by obtaining a copy of the target device and uploading a simplified firmware, where finding the timing for fault injection is trivial. This allows the attacker to fix the timing parameter in order to reduce the search space by an order of magnitude. To then find effective intensity parameters, the attacker simply performs a sweep over the possible configurations for as many times as desired. When enough data is collected about different intensities and the observed behavior of the target, it becomes clear which combination of parameters provides the most consistent results.

After finding effective parameters for the intensity by simplifying the timing with a modified firmware, these parameters can be fixed and the attacker can start trying to find an attack timing that will work during the attack phase. This can be achieved in a multitude of ways, such as gathering timing information using performance counters or by measuring side channels such as power usage and EM emanations; section 3.1.3 provides a summary of such methods and section 5.1.3 describes an attempt that was made at developing a RISC-V specific method within this research. However, given the state of previous work [31] together with the relative technical difficulty of applying these methods, this step is currently ignored.

4.1.3 Attack Phase

During the attack phase, the parameters obtained during the profiling phase(s) are applied to the original TEE firmware and victim application running on the target device. The attacker application is installed in the TEE, which will output a trigger signal and attempt to dump the victim's memory, where it stores its secret key.

Figure 4.2 shows a sequence diagram of the attacker app attempting to read out the victim app's secret key during the attack phase. If a fault was successfully injected, such that the PMP configuration got corrupted to a degree that the malicious read operation from the attacker is allowed by the PMP unit, the attacker will read and print out the victim's secret key. If the fault injection failed and the PMP configuration remained as intended, the hardware will trigger an interrupt signaling an illegal memory access (`MCAUSE=0x5`), allowing the TEE monitor to undertake action and block any further execution of the attacker app.

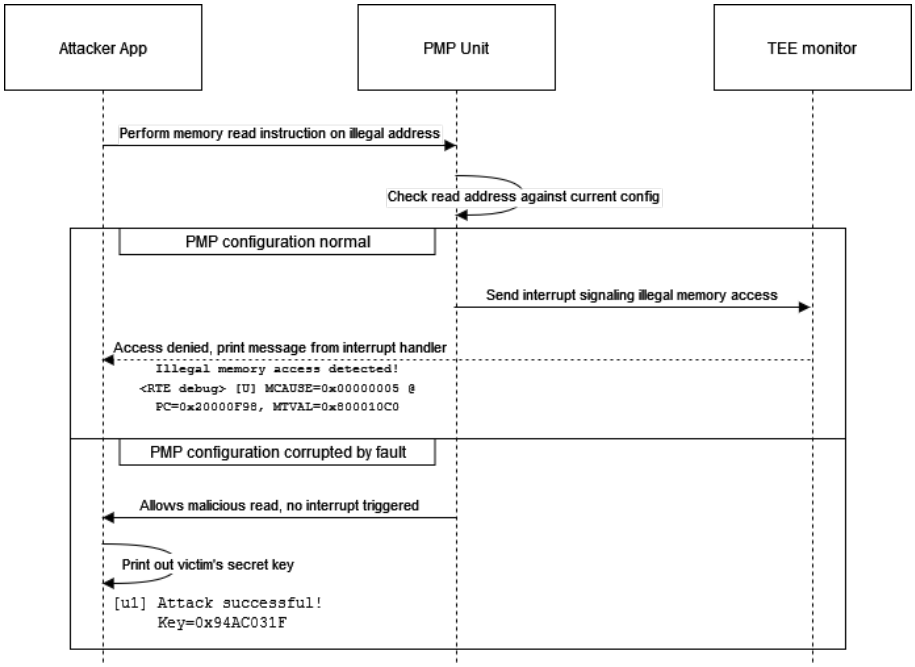


FIGURE 4.2: Sequence Diagram of an Attempted Malicious Read Operation

4.1.4 Clock Glitching

There are numerous ways of injecting faults for the purpose of the described attack scheme. All of the ones described in section 2.1.1 could theoretically be effective on the hardware that will be targeted in this work. However, considering that investigating different fault injection methods is not the purpose here, only one method is elected for the upcoming experiments: clock glitching. The reason for using clock glitching is simply its cost efficiency and ease of use, while still offering some precision.

As briefly explained previously, clock glitching is a method of fault injection that involves an attacker generating their own clock signal and feeding it into the target processor. By doing so, it opens up the possibility of the attacker modifying the clock signal such that the target's timing requirements are violated, at which point undefined behavior may start to occur in the target, leading to hardware faults.

More specifically, the intention of an attacker in this case would be to corrupt the contents of critical registers by introducing a clock cycle with a duration that is just short enough to violate the setup or hold time of those registers, while ideally not affecting any other registers as this may cause unwanted behavior. Since the critical path to each flip-flop in a digital circuit will (slightly) differ from the critical paths to other flip-flops, it is theoretically possible to target each one with a specific glitch shape (or intensity). However, it may not be possible to target each one separately, possibly leading to unwanted side effects.

4.2 TEE Design

In this research, a PoC-TEE is used and adapted to enforce separation of user-level applications by utilizing RISC-V's Physical Memory Protection extension and privilege modes (please refer to section 2.2.2 for background on these subjects). The PoC-TEE serves as a case study to motivate the investigation of certain instructions that are critical to its operation.

4.2.1 Global Overview

Figure 4.3 shows a flowchart of how the PoC-TEE operates and where the attack vector lies in its operation. The goal of this design is to simplify the concept of a TEE as much as possible, while still retaining all of the guarantees provided by a 'real' TEE, most importantly the separation in memory between applications, using the same mechanisms: 2 privilege modes and the PMP extension.

The PoC-TEE's primary behavior is that it performs context switches between two applications: the victim app and the attacker app; with each context switch, the PMP configuration is switched such that the app running next in sequence cannot access the memory of the previous app. More specifically, after a reset the PoC-TEE starts by performing the boot process and preparing the PMP configuration to run the victim app. When the victim app is done running its encryption, it executes an `ecall` instruction to trigger an exception, which the monitor handles by performing a context switch to the other application, in this case the attacker app. The attacker app will then send out a trigger signal on one of the GPIO pins shortly before requesting another context switch from the monitor; the trigger signal tells the attacker's external hardware that something critical is about to happen (a context switch) and it should start counting cycles until it is the right time to inject a fault. During the context switch, the attacker aims to glitch the exact clock cycle when the PMP configuration CSR is being set, possibly leading to a corrupted configuration. When execution returns to the attacker app, it attempts to read out the victim's memory, which is shown in figure 4.2.

However, this is just the ideal case for the attacker. For example, it might occur that the clock glitch does not cause a fault to be injected, causing the attacker's malicious read operation to be caught by the TEE. Another scenario might be that the clock glitch causes too many undesired side effects and causes other parts of the software to malfunction. This is solved by simply resetting the target and executing the attack again; because of the probabilistic nature of FI attacks, it may require many tries before an attack is successful.

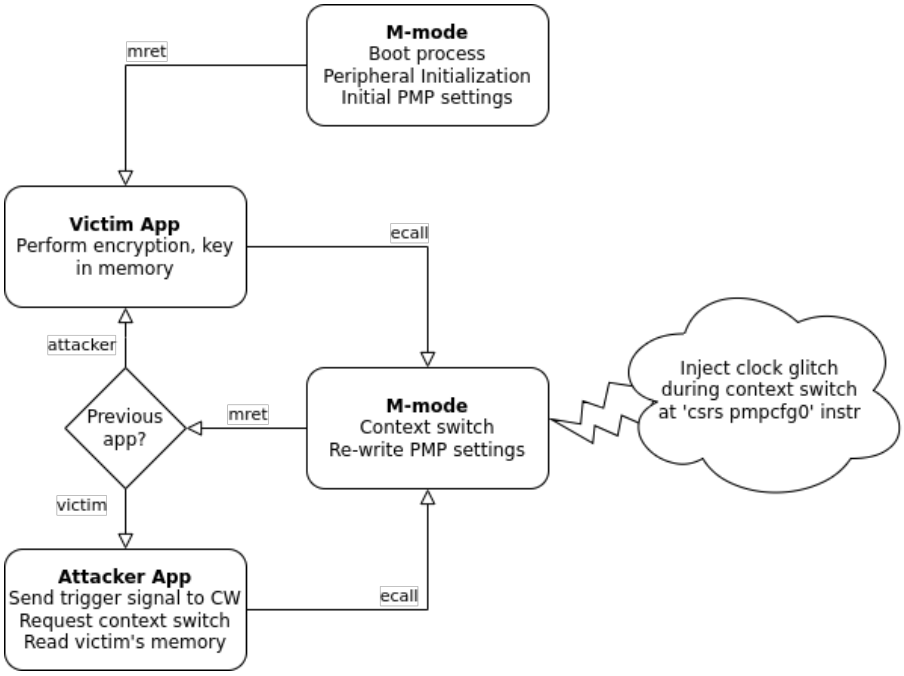


FIGURE 4.3: PoC-TEE Flow Chart

4.2.2 Memory Layout

Figure 4.4 shows the memory layout of the PoC-TEE firmware, showing exactly which parts of memory are protected and to what degree. The figure shows the memory layout of a typical SoC used in embedded systems, having two separate physical memories: a Read-Only Memory (ROM) and a Random Access Memory (RAM). The RAM is commonly used to store runtime data that is dynamically updated by the software, while the ROM is typically chosen to contain read-only sections of memory such as code, as these sections are not expected to be written to; as a useful byproduct, this introduces some physical protection against tampering with the data stored in the ROM.

For this TEE, it has been chosen to disable any memory protection on the ROM, as it just contains the code (`.text` sections) and the read-only data (`.rodata` sections). Given the attacker model, where it is specified that the TEE's code must be open source, this is already considered to be known by the attacker. Furthermore, as mentioned previously the ROM protects against write operations by its nature, which means tampering with these sections is not considered a threat. However, in the case of a system that does not have a ROM, these sections should at least be protected against write operations.

The runtime data of each process (TEE monitor and each user app) is stored in the RAM, as it needs to be dynamically read and written to. Because the content of this part of memory is not known at compile-time and could be of interest to an attacking party, since it might for example contain an encryption key, it is in need of memory protection. First of all, to eliminate the possibility of tampering with the TEE monitor it is configured to never be accessible in any way, except when running in the highest privilege mode, which is only possible when the monitor's own code is being run. By default, the regions assigned to each user application are also configured to be inaccessible, except when the monitor is about to switch the context to the app in question, at which point it configures the PMP to allow read and write operations to that app's runtime memory.

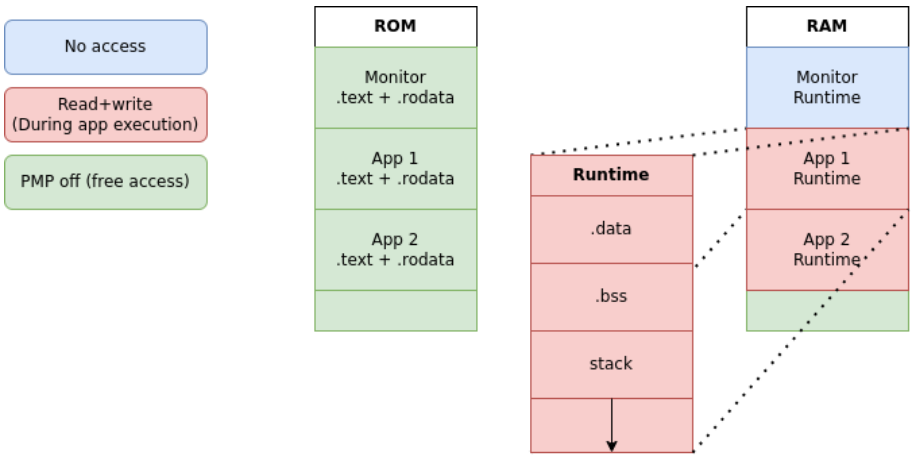


FIGURE 4.4: PoC-TEE Memory Layout

4.2.3 PMP Configuration and Target Instructions

When a RISC-V hardware thread that supports the PMP extension performs any action that requires memory access, by default, the hardware-defined PMP unit implicitly verifies whether the address that is being accessed is authorized according to the current privilege level and PMP configuration. This includes the execution of simple memory access instructions, such as read and write operations, but also the fetching of every instruction that is going to be executed.

The memory protection regions and their specific rules are configured solely by a set of CSRs: the `pmpcfg0-pmpcfg3` CSRs, which control the rules for each region (4 8-bit values corresponding to a region per CSR) and the `pmpaddr0-pmpaddr15` CSRs, which define the address range of each region. The two basic instructions that are able to write to these regions are the `csrw` and `csrs` instructions; `csrw` allows the programmer to write a value from a general purpose register to the specific CSR, while `csrs` gives the possibility of specifying a mask for the CSR write operation.

Even though there exist various, slightly different, alternatives to both of these instructions, such as ones that read the old value at the same time that they write the new value, the differences are too slight and the number of instruction variants too numerous to draw a meaningful comparison. Therefore, it is concluded that the `csrw` and `csrs` instructions should both be considered in the experiments, as they are both commonly used in configuring the memory protection offered by a TEE on RISC-V. However, their sub-variations are not considered.

Chapter 5

Experimental Setups

This research performs the profiling phase for glitch intensity, as described in the previous chapter, as a means to evaluate a core's susceptibility to FI attacks using clock glitching. The experimental setup is kept as similar as possible between testing the 2 different cores. For each tested core - the E31 and the Ibex - 2 different instructions are profiled. The setup that is required for these experiments, as well as relevant details about the tested systems are specified in this chapter.

5.1 Fault Injection Setup

Figure 5.1 shows a block diagram that provides a high level overview of the setup used to perform all experiments on the E31 core. Additionally, Figure 5.2 provides a physical overview of the experimental setup. There are three main components to the setup:

- **The target chip:** in this experiment there is an FE310 RISC-V board, containing the E31 core, mounted on the ChipWhisperer CW308 target board, which allows for convenient access to the FE310's pins, including its clock input in order to feed it a clock glitch.
- **The glitch generator:** in this case the ChipWhisperer CWLite is used to generate a clock signal for the target chip, inserting a clock glitch when a trigger signal is received. Additionally, it controls the reset input of the target chip, which provides more control during the experiments.
- **The glitch controller:** controls the glitch generator by specifying when to reset the target and the glitch parameters (intensity and timing). It also controls the target in this case by sending it data over UART. Lastly, it observes the UART output from the target, from which it is able to roughly determine the result of each fault injection attempt. In this case a Raspberry Pi is used to host a Jupyter server where a script can be executed that controls the process of systematically repeating fault injection attempts.

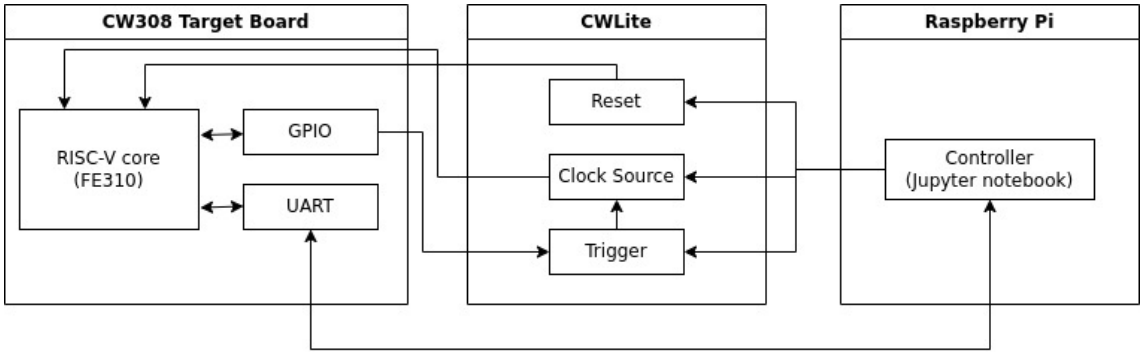


FIGURE 5.1: Block Diagram of Devices in Experimental Setup

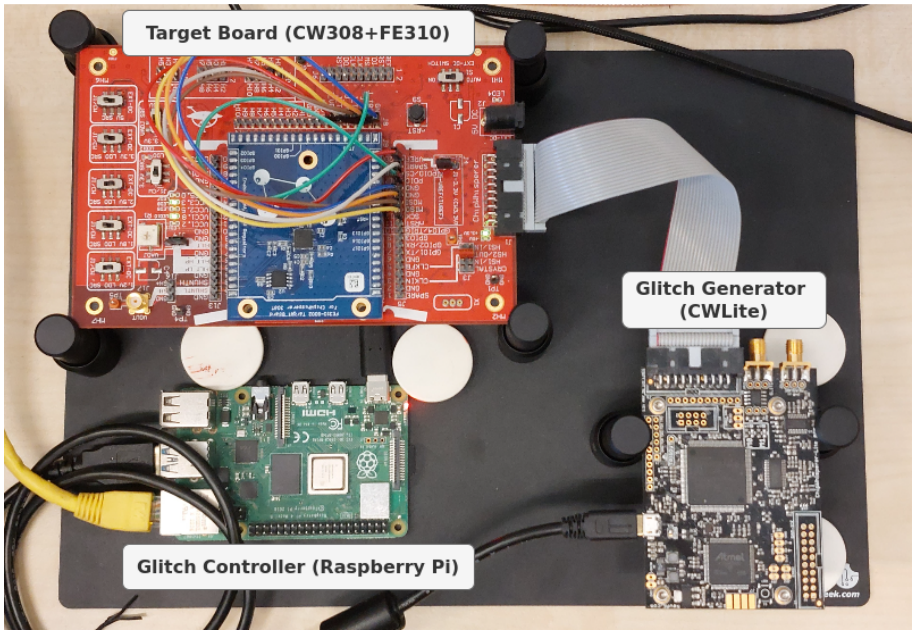


FIGURE 5.2: Physical Overview of Experimental Setup for FE310

The setup is kept largely the same when considering the Ibex core. However, where the CW308+FE310 target was used to assess the E31 core, the CW305 is now used. The CW305 hosts a Xilinx Artix-7 FPGA, which is used in this case to deploy an Ibex core along with peripherals to interface with it during runtime. Figure 5.3 shows a physical overview of the experimental setup when assessing the Ibex core.

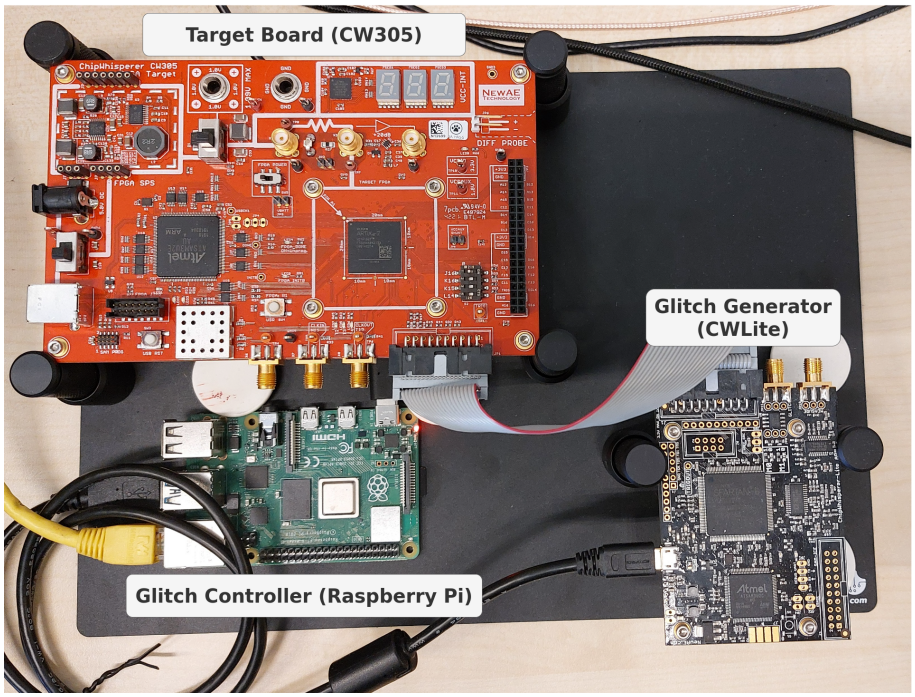


FIGURE 5.3: Physical Overview of Experimental Setup for Ibex

5.1.1 Clock Glitch in Effect

Figure 5.4 shows a capture of the clock glitch taking effect in real time during one of the experiments. For illustration purposes, the glitch generator is configured to output a glitch with a delay of 3 clock cycles, a width of 25% and an offset of 25%. The timeline is as follows: at time `t1` the attacker application makes the target board output a trigger signal to the glitch generator, after which at `t2` the glitch generator has registered the trigger and starts counting down the delay cycles until `t3`, where the glitch should be inserted into the clock signal. In figure 5.5 the glitched clock cycle is shown in more detail; it can be observed that the glitch takes effect at `t4`, after 25% of a clock period from the start of the affected clock cycle and also has a duration of 25% of a clock period, continuing with a normal clock signal after `t5`.

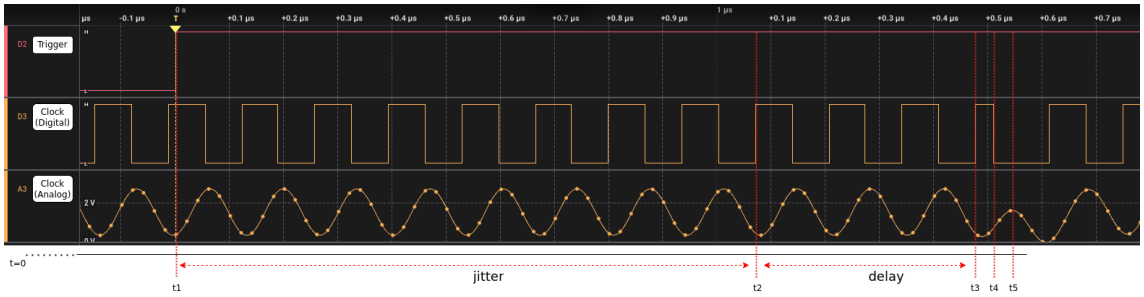


FIGURE 5.4: Clock Glitch in Effect During Experiment

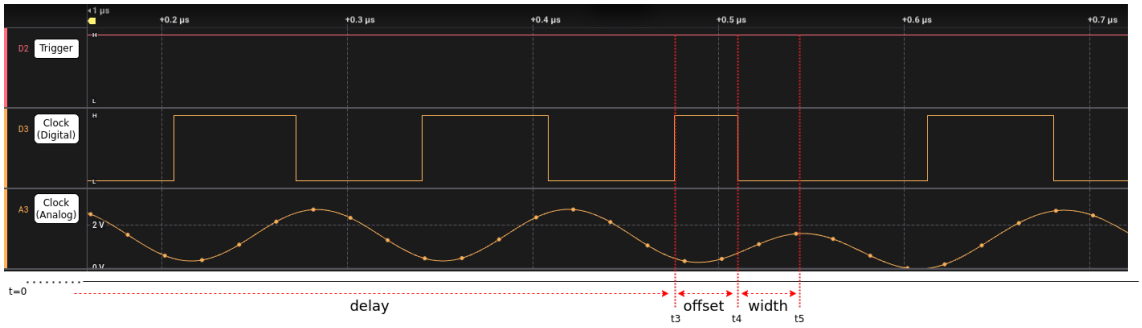


FIGURE 5.5: Glitched Clock Cycle During Experiment in Detail

5.1.2 Profiling I - Glitch Intensity

After determining the target instruction, the main challenge of successfully performing a fault injection attack is to determine the right glitch parameters. For a clock glitching attack, the parameters can roughly be subdivided into two categories: intensity and timing, both of which are covered separately below. The intensity of a clock glitch is determined by its width and offset relative to the clock period, while the timing is solely measured by the number of clock cycles that the glitch generator waits to insert the glitch after receiving a trigger signal. For a more in-depth explanation of the parameters related to clock glitching attacks (and fault injection in general), please refer to section 4.1.4. Algorithm 1 shows the algorithm used by the glitch controller to profile a target for the appropriate intensity.

As shown in the algorithm, each possible combination of width and offset is tested 100 times in order to get a sufficient number of samples. Each experiment is logged as being either a `success`, a `reset` or `normal`, following these definitions:

- **Success:** a CSR write was observably skipped or caused an undefined value to be written to the CSR, while resuming normal execution.
- **Reset:** a fault was injected, but caused excessive effects leading to a non-responsive target or the target otherwise diverted from normal execution, needing to be reset.
- **Normal:** The glitch was either too weak to inject a fault or the propagating effects of the fault were not observable. The target resumed normal execution.

Algorithm 1 Algorithm executed by the glitch controller

```
delay ← 17                                   ▷ Fixed 'Known-to-work' delay to profile for intensity only
for i ∈ {1, 2 ... 100} do
    for width ∈ {−45, −44 ... 45} \ {0} do
        for offset ∈ {−45, −44 ... 45} \ {0} do
            configureGlitch(width, offset, delay)   ▷ Passes config to glitch generator
            prepGlitch()    ▷ Signals glitch generator to perform glitch at next trigger
            capture()                                ▷ Waits until glitch has been performed
            out ← readUART()                                  ▷ read incoming UART buffer
            if 'Success' ∈ out then
                logSuccess()                         ▷ Log fault injection with desired effect
                resetTarget()                  ▷ Reset the target board for next injection
            else if 'Normal' ∈ out then
                logNormal()              ▷ Log fault injection without propagating effect
            else
                logReset()                ▷ Log fault injection resulting in a crash/reset
                resetTarget()
            end if
        end for
    end for
end for
```

5.1.3 Profiling II - Glitch Timing

Now that effective intensity parameters are known, the timing of the glitch insertion is the only parameter that still needs to be discovered. A naive way of determining the timing is to fix the intensity parameters to a set that is known to be the most effective, after which sweeping the `delay` parameter until there is a successful attack. However, in the majority of real attack scenarios there is a rather significant (and often unknown) time between the attacker application signaling a trigger and the target instruction being executed. This often causes a naive parameter sweep to be unfeasible due to the explosion in search space, considering that a single experiment in a fault injection campaign might take anywhere from a few 100 milliseconds to a few seconds.

However, there are more time-efficient ways of determining the correct timing that can be applied here. The most notable of which is 'template matching', which is used in [44] and [4] and consists of using side-channel analysis to record a trace of during a known critical section, which is then used to match against a real-time trace during the target's execution. However, since these methods are not applied in any previous works involving RISC-V, applying one of them is not considered to be in the scope of this research. Instead, a trigger signal is generated internally by the TEE monitor, similarly to Nashimoto's [31] recent work.

Even though profiling for the timing of the glitch is purposefully left out of this work for reasons mentioned above, it should be noted that an attempt was made at introducing a novel method for obtaining a timing estimate using RISC-V specific performance counters. Within a RISC-V core there exists a CSR called `mcycle`, which can be set to an arbitrary value and counts up by 1 for every clock cycle. During the profiling phase, it is theoretically possible to start the timer (i.e. setting it to `0`) right after the attacker app is supposed to output the trigger signal and measuring it at the injection point in the TEE monitor. In theory, this would give a timing estimate that is accurate enough to find the actual timing

by 'brute force'. However, some yet unknown complications (possibly due to its instruction cache) with the FE310 SoC cause the timing to be extremely jittery. This jitter is far too large (in the range of thousands of clock cycles) to perform a feasible brute force search around the estimate. However, this method might be applicable to other SoC's, which is the reason for including these thoughts here.

5.2 Attacking the E31

The first experiment is conducted on SiFive's E31 core, which is a hard RISC-V core suitable for use in embedded systems. It is not documented [40] to have any countermeasures against hardware attacks and serves as a baseline to compare with the second experiment (see chapter 5.3), where the same attack is evaluated against a different RISC-V core.

5.2.1 Hardware

The E31 core [39], the RISC-V core used on the FE310 [40] development board, has a 5-stage pipeline consisting of the following stages: instruction fetch, instruction decode/register fetch, execute, data memory access and register writeback. The pipeline has a maximum throughput of 1 instruction per clock cycle, but in some cases it might incur a penalty of a few clock cycles due to so-called hazards, which occur when a subsequent instruction requires the result of the previous instruction, thus requiring the pipeline to stall execution until the required instruction is completed. Other cases where a penalty might be incurred is when a branch is mispredicted or when there is a cache miss on a memory reading operation. It should be noted that branching and memory-related instructions generally tend to incur a small penalty, even if there is a cache hit or correct branch predictions. Therefore, as an attacker, it is especially interesting to attack at these points, as the prolonged operation of a certain instruction gives more fine-grained control and a larger window to attack. Additionally, since no other operations are taking place at the same time during a stall or pipeline flush, there is a reduced chance of causing unwanted side effects when injecting a fault during that time.

In the E31 core specifically, memory access instructions have an additional latency of 2 or 3 cycles depending on the specific instruction; mispredicted branches incur a 3 cycle penalty, with correctly predicted branches not incurring any penalty. This makes memory access instructions and mispredicted branches an interesting target for FI attacks in general. However, what stands out in this case is the requirement to flush the pipeline on a CSR write operation, which effectively causes all subsequent instructions to be stalled until the CSR is written to. This means that when a CSR is written to, which is often a critical part of trusted execution (when setting the PMP configuration), there is 5 cycle window for an attacker to exploit, where only the CSR write instruction is being executed. Therefore, it follows that this experiment on the E31 also makes use of this window.

Another concern when using a clock glitching method for an FI attack is the clock domains that exist, as targeting a different clock domain will allow the attacker to target different parts of the system. In the case of the FE310 package, there are 2 clock domains: the always-on (AON) domain and the mostly-off (MOFF) domain. Both can be supplied with an external clock, technically allowing an attacker to use clock glitching to target either one. For example, the AON domain contains the power management unit, which might be targeted to induce an error in the power supply (i.e. a possible voltage glitch). However, in this research, one of the main points is to evaluate a generic attack on TEEs for RISC-V, meaning that attacking an implementation-specific detail, which does not even

contain the E31 core (the E31 core exists in the MOFF domain), would defeat the purpose. Therefore, the focus will lie on attacking the domain where the E31 core resides, focusing on targets that are common between RISC-V cores.

In conclusion, the CSR writing instructions that should be targeted according to the attack scheme seem to be a promising target on the E31 because of their behavior in the pipeline. Since the core is not documented to implement any FI-specific security features, it is expected that CSR writing instructions on the E31 will be quite vulnerable to attack by clock glitching.

5.2.2 Firmware

The goal of this experiment is to determine how effectively the PoC-TEE could be attacked by an attacker. For this purpose, an altered version of the PoC-TEE firmware is used, since the attack model dictates that the attacker may have their own copy of the hardware target in question, as well as the firmware's source code.

When trying to find effective glitch parameters as an attacker, it is often a good idea to eliminate parameters from a sweep where possible. Since an attacker is often looking for multiple glitch parameters related to the shape and timing of the glitch, the search space easily explodes and becomes unmanageable.

To this end, a simplified firmware is used that eliminates the timing parameter by placing the target instruction shortly after triggering the attack. By doing so, the attacker can immediately search for an effective glitch intensity (i.e. width and offset) without having to consider timing, since the glitch is guaranteed to hit the target instruction. Listing 5.1 shows the critical section of the firmware used to extract the glitch intensity parameters when targeting the `csrw` instruction. In this case, the critical section starts at the point when the glitch generator is triggered and ends when the trigger signal is pulled back to 0. During the time that the trigger is high, the processor has to write a predefined value to 8 different CSRs that influence the PMP settings. It should be noted that communication between the target and the controller is ongoing outside of this critical section in order to control the target and observe its behavior for data collection, this is intentionally left out of the code listing.

```
1  jal     trigger_high                    # sends the trigger signal
2  li      a3, 0xabcdef                    # load predefined value
3  csrw    pmpaddr0,a3                     # start writing to each CSR
4  csrw    pmpaddr1,a3
5  csrw    pmpaddr2,a3
6  csrw    pmpaddr3,a3                     # <-- INJECT FAULT
7  csrw    pmpaddr4,a3
8  csrw    pmpaddr5,a3
9  csrw    pmpaddr6,a3
10 csrw    pmpaddr7,a3
11 jal     trigger_low                     # no longer send trigger
```

Listing 5.1: Firmware Used for Extracting Glitch Intensity Parameters on FE310

5.3 Attacking the Ibex

The second experiment is conducted on lowRISC's Ibex core [28], which is an open source RISC-V core. In this experiment, the effects of clock glitching are investigated similarly to the experiment described in section 5.2, such that a comparison can be drawn.

5.3.1 SoC Structure

Figure 5.6 shows a global overview of the SoC being targeted during this experiment. It is deployed as a soft core on the FPGA of the CW305 target board and consists of an Ibex core, which is described in more detail below, and SRAM along with peripherals such as:

- **GPIO** is needed to let the attacker application signal the glitch generator when to 'start' attacking. Similarly to the previous experiment, this is done to simplify the experiment to the part of interest. In a 'real' scenario, there are a multitude of ways to acquire an actual trigger, as described in section 3.1.3.
- **UART** is used to communicate with the target, either signaling it to start executing critical code or analysing its output to determine normal, reset or success behavior.
- The **debug module** is needed to upload the firmware to the RAM and to reset the core when it experiences an observable effect caused by FI.

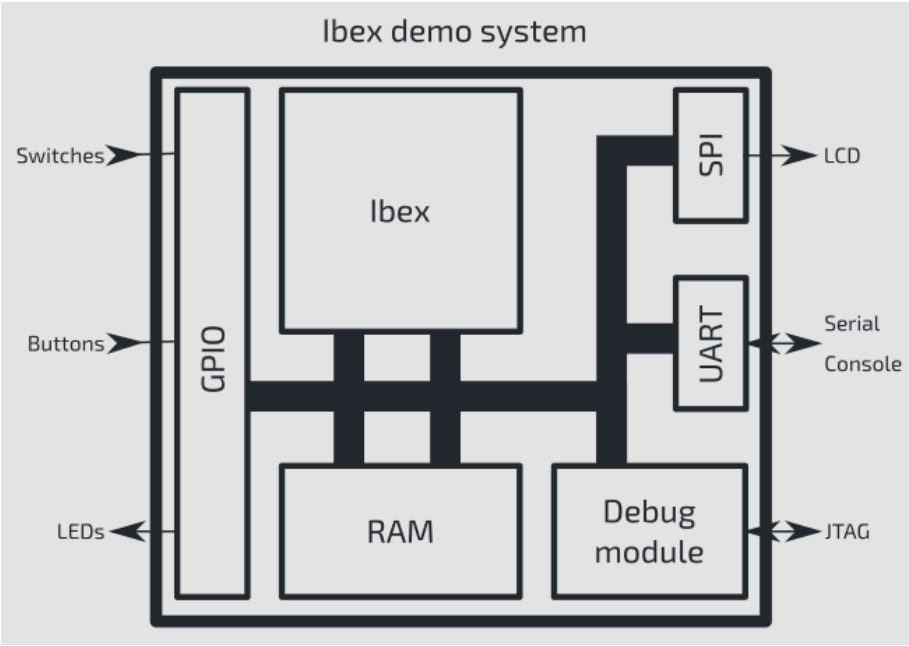


FIGURE 5.6: Block Diagram of the Used Ibex System [27]

5.3.2 Ibex Core

The Ibex core [28] has a 2-stage pipeline consisting of an instruction fetch and an instruction decode and execute stage. Similarly to the E31, it has a theoretical maximum throughput of 1 instruction per clock cycle, but might be stalled because of exceptional events, such as branching instructions, memory access instructions, or costly operations such as integer division and multiplication. Additionally, with the Ibex it is again the case that CSR write instructions incur a pipeline flush [26], meaning that it is also a promising target for clock glitching. However, given that a CSR write is only in the pipeline of the Ibex for 2 cycles instead of 5 with the E31, it might be harder to successfully inject a fault, as this offers a reduced window of control for the attacker.

Furthermore, with the Ibex there is the possibility of enabling hardware-based security features that are known to mitigate FI attacks, more information on some of these features can be found in section 3.2.1. Naturally, enabling this during the Ibex's synthesis would greatly reduce the effectiveness of FI attacks, but for the purpose of this research, security features have not been enabled. In this case, the Ibex is not documented to have any security features that counteract FI attacks, similarly to the E31.

Finally, the Ibex, when deployed in the previously described SoC, only has 1 clock domain as opposed to the FE310's 2 clock domains. This means that when it comes to clock glitching, there is no choice to be made in terms of locality. However, as the Ibex is open source, it does offer a possible additional degree of knowledge to the attacker in comparison to the E31's closed source ASIC implementation. Since all of Ibex's implementation details are technically known, the attacker could for example analyze critical paths in the logic in order to more specifically target certain flip-flops with timing violations, for example those that make up the CSRs.

In conclusion, the Ibex core seems to offer a bit less freedom for an FI attacker in comparison with the E31, mainly due to its shorter pipeline. However, its open source implementation does provide a potential attacker with additional information.

5.3.3 Firmware

The firmware for this experiment, shown in listing 5.2 has been slightly altered to account for microarchitectural differences, while retaining the essence of the first experiment (please refer to section 5.2.2). As mentioned earlier, the default configuration of the Ibex core in the SoC does not implement the PMP extension. However, since the attack targets CSR access instructions that configure the PMP unit, rather than directly targeting the PMP unit, the experiment still gives the same insight when targeting other CSRs.

To this end, the `mhpmcounter3..10` CSRs have been chosen for this experiment, since writing values to these registers does not have an inherent effect on the core or the firmware running on it. In preparation for the critical section, the `mcountinhibit` register must be set appropriately, such that actual usage (incrementation) of the performance counters is inhibited and does not influence the result of the experiment.

```
1  # Before glitching inhibit all counter CSRs
2  # Allowing them to be used as general purpose registers
3  li      a3, 0xffffffff
4  csrw    mcountinhibit,a3
5
6  jal     trigger_high                    # sends the trigger signal
7  li      a3, 0xabcdef                    # load predefined value
8  csrw    mhpmcounter3,a3                 # start writing to each CSR
9  csrw    mhpmcounter4,a3
10 csrw    mhpmcounter5,a3
11 csrw    mhpmcounter6,a3                 # <-- INJECT FAULT
12 csrw    mhpmcounter7,a3
13 csrw    mhpmcounter8,a3
14 csrw    mhpmcounter9,a3
15 csrw    mhpmcounter10,a3
16 jal     trigger_low                     # no longer send trigger
```

Listing 5.2: Firmware Used for Extracting Glitch Intensity Parameters on Ibex

Chapter 6

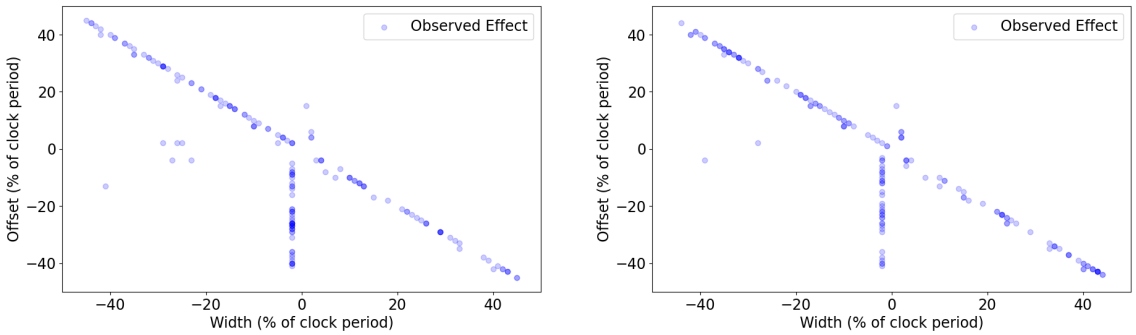
Results

Following from the previous chapter, where the experiments are described for both cores used in this study, the steps for the first profiling phase are followed to find how susceptible each core is to clock glitching attacks. To summarize: the experiment is conducted on both cores, targeting the same instructions and using minimally altered firmware. Each parameter combination is tried 100 times in order to obtain sufficient amounts of data, since fault injection attacks are known to be a stochastic process.

6.1 Susceptibility of the E31 Core

Figure 6.1a and 6.1b give an overview of effective width and offset parameter combinations for the `csrs` and `csrw` target instructions respectively in the form of a scatterplot. Each shaded dot in the scatter plot signifies that at least one effectful fault injection (either a success or a reset) was observed at that combination of width and offset. A darker shade already gives a hint towards the probability of causing effects. Later in this section, figures 6.2 and 6.3 illustrate more fully how probabilities for both types of effects are distributed.

The scatter plots show two clear bands of effects in both target instructions, along with a few outliers. It should be noted that the results look quite similar when comparing the `csrs` and `csrw` instructions, which can be attributed to the fact that both instructions essentially deal with the same basic operation: writing a value to a CSR. Thus, they likely follow a very similar data path with similar critical paths, leading to a similar susceptibility to timing violations caused by clock glitches. The minor differences, mainly with respect to the outliers, can be attributed to the minor differences in data path between both instructions, as `csrw` writes a value from a register to a CSR, while `csrs` applies a mask as well.



(A) Profiling for `csrs`

(B) Profiling for `csrw`

FIGURE 6.1: Profiling Scatterplots for E31

Figure 6.2 shows a more detailed representation of the density of effects for both parameters when profiling the `csrs` instruction, separating the 2 categories of effects: successes and resets. The graph on the left fixes the offset parameter, showing the probability of successes and resets for different widths, while the graph on the right fixes the width.

From this figure, as well as figure 6.3, the probabilistic nature of FI attacks becomes clear, since even the most successful parameter combinations do not provide any guarantee of a successful effect. It can also be seen that most effects are concentrated around a specific width, with successful offsets being more uniformly distributed. Going from these graphs, the attacker should fix the width between 0 and -5 and the offset between -25 and -30 when going to the attack phase of the attack scheme (refer to section 4.1 for more details), as this will maximize the success chance.

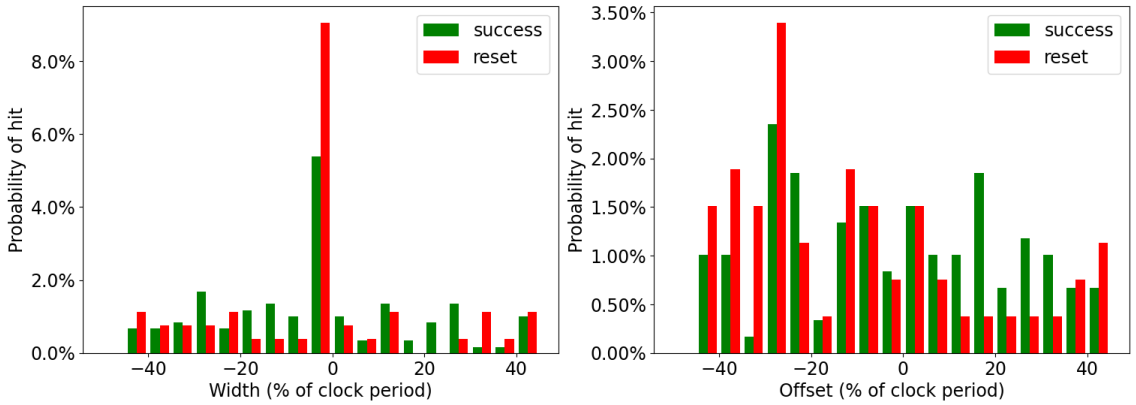


FIGURE 6.2: Probabilistic Representation of Profiling for `csrs` on E31

Figure 6.3 again shows a more detailed, probabilistic view of the first profiling step, in this case for the `csrw` instruction, similarly to figure 6.2. There are some small differences between the two instructions though. For example, the distribution of effects seems to be more uniformly distributed `csrw`, having a less pronounced peaks of effects. Because of this, there are a few areas of interest that an attacker could fix their parameters on; for example, choosing a very negative width and very positive offset gives quite a high chance success, which is also partially visible in the top-left corner of figure 6.1b. The same could be said for the exact opposite corner, as there seems to be a similar concentration of effects there.

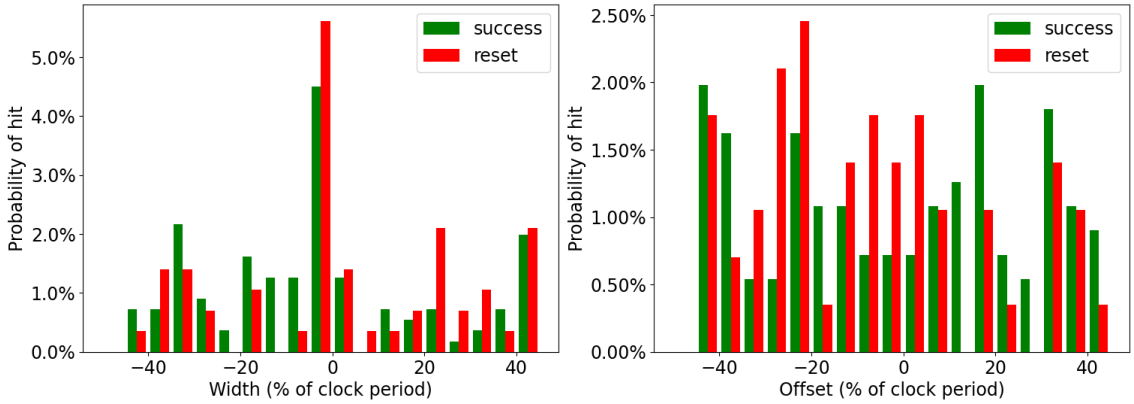
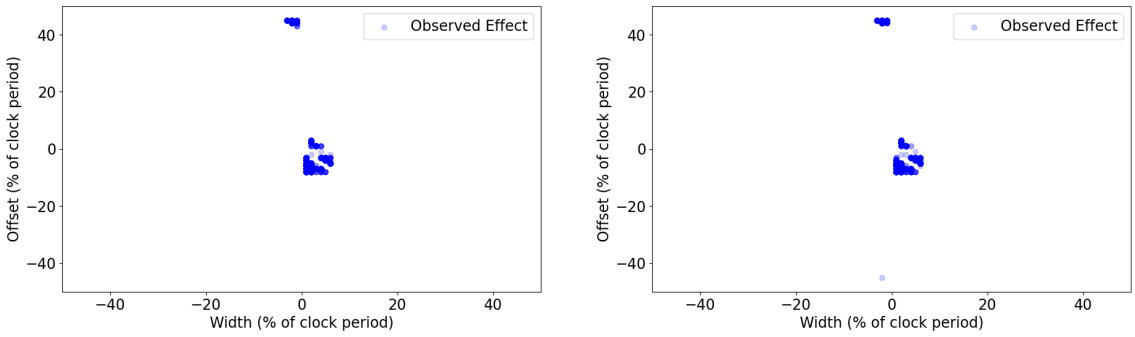


FIGURE 6.3: Probabilistic Representation of Profiling for `csrw` on E31

6.2 Susceptibility of the Ibex Core

Figure 6.4a and 6.4b show an overview of the effects of all width and offset parameter combinations for both target instructions, similarly to figure 6.1a and 6.1b, which illustrate the same concept for the E31. Again, each shaded blue dot signifies the occurrence of at least one effect (either success or reset) observed at that width and offset combination, with a darker shade indicating a higher rate of observed effects. Figures 6.5 and 6.6 show a more detailed representation of how both types of effects are distributed across the parameters, similarly to figures 6.2 and 6.3 for the E31.

The graphs for both target instructions are quite similar again, which is also seen during the previous experiments on the E31 core. There are two distinct areas of parameter combinations in both graphs that seem to yield the most effects, which likely corresponds to the parts of the data path where both instructions are similar. However, this time the `csrw` seems to offer an outlier where the `csrs` does not, likely owing to a minute difference in their data paths where `csrw` exposes a slight susceptibility to FI.



(A) Profiling for `csrs`

(B) Profiling for `csrw`

FIGURE 6.4: Profiling Scatterplots for Ibex

What stands out from figure 6.5 in comparison with the experiments on the E31 core, is that there have been no successes when using the Ibex. A possible explanation for this could lie in its microarchitectural details (see section 5.3.2 for more details). For example, the reduced pipeline length restricts the attacker's ability to target specific parts of the target instruction's execution, possibly leading to unwanted side effects.

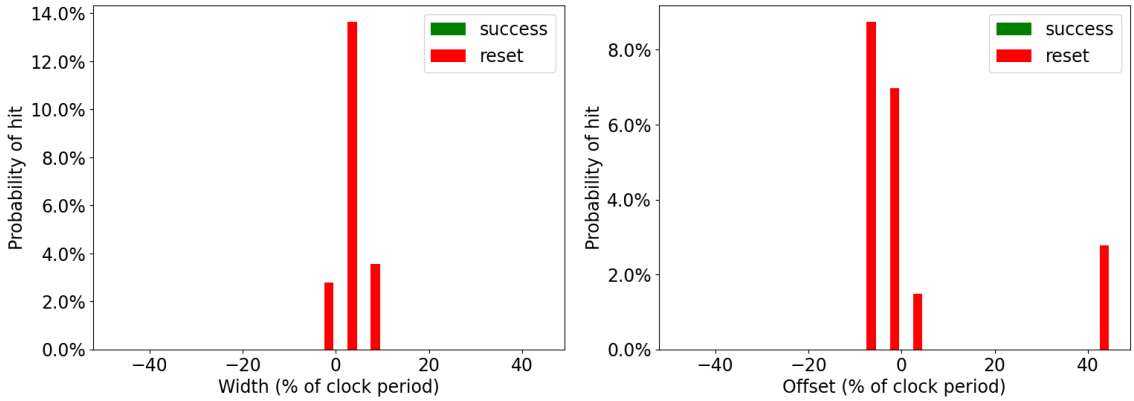


FIGURE 6.5: Probabilistic Representation of Profiling for `csrs` on Ibex

Figure 6.6 shows the results for the `csrw` instruction, which paints a similar picture to what is seen for the `csrs` instruction. There are only resets, but no successes, meaning that there are likely too many unwanted side effects occurring.

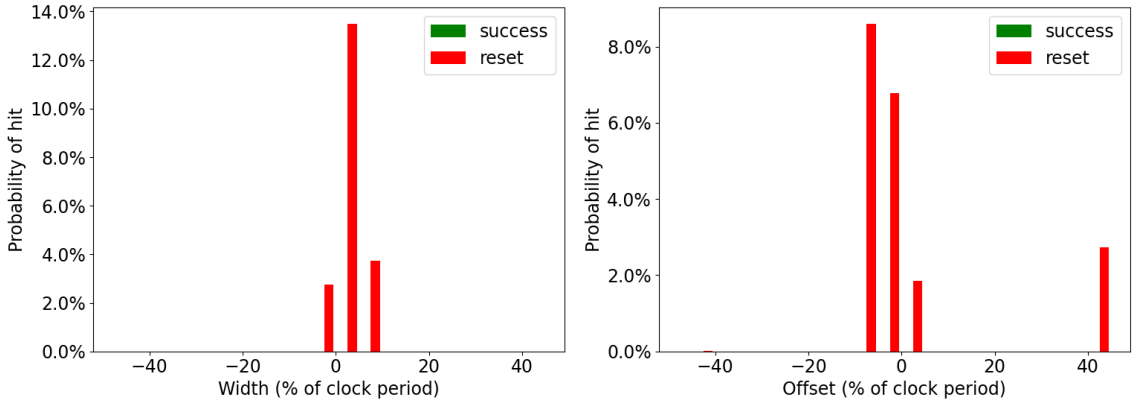


FIGURE 6.6: Probabilistic Representation of Profiling for `csrw` on Ibex

Chapter 7

Discussion

In this thesis it is shown that performing the same fault injection attack on nearly identical firmware running on different hardware targets gives vastly different results. Supporting the hypothesis and indicating that in the scenario of a physical attack, a TEE is at most as secure as the hardware that it is running on.

7.1 Effective Attack Vectors

The first subquestion asks which attack vectors are the most effective in attacking a TEE on a RISC-V core. Technically, targeting almost any instruction with a fault injection attack has the possibility of propagating the desired effect. However, during the development of the PoC-TEE case study, it became clear which type of instructions would be most promising to target during an attack.

For example, an attacker could try to corrupt a branching instruction such that the program counter skips over a security-critical portion of code. However, this would require the injected fault to propagate in an extremely improbable manner, eventually corrupting the program counter to the exact desired position.

A more direct target would be the specific RISC-V instructions that deal with access to the control and status registers (CSRs). As the name implies, these special registers that exist in many RISC-V cores give direct control over parts of the core and contain valuable status information. Write operations to the CSRs are often done to configure security-critical parts of the system, for example the protection of memory zones in a TEE. When these instructions are skipped, the configuration is not updated and an application, possibly that of the attacker, will have access to memory that it is not supposed to have.

As is shown in the results, there does not seem to be a major difference between targeting different instructions on the same core that can be used to configure the CSRs. Since all instructions that deal with writing to CSRs in some way likely follow a similar critical path in the hardware, they are similarly affected by timing violations caused by clock glitches. There are some exceptions to the similarity in both experiments, which likely account for the slight difference in the data path of each instruction.

7.2 Difference Between Cores

The second subquestion addresses the influence that different microarchitectures have on the effects observed from injecting faults in RISC-V cores. From the conducted experiments, it seems to be the case that there is a vast difference between different cores running similar firmware. However, some similar patterns do arise between the different instruction targets on the different cores.

The most notable difference arises in the effectiveness of glitch parameters, or the shape of the glitch. While the FE310 is affected across a broad range of parameters, with successes occurring throughout, the Ibex system does not see any of the attacker's desired effects, only undesired effects or normal behavior. This implies that microarchitectural differences indeed play a large role in the effectiveness of FI attacks involving clock glitching. This difference could be attributed (in part) to the different pipeline structure of each core, a more detailed description of which can be found in sections 5.2.1 and 5.3.2 for the E31 and Ibex respectively. It is theorized that the E31, having a 5-stage pipeline as opposed to the Ibex's 2-stage pipeline, offers more fine-grained control over which part of the instruction's execution is affected by the clock glitch; since each instruction is executed over 5 or more clock cycles (depending on stalls), the attacker may choose between which stage(s) to attack specifically.

However, the lack of precision when attacking the Ibex does give some insights into future steps that could be taken. Referring back to section 2.1.1, where different FI methods are discussed, the lack of precision offered by the Ibex's pipeline could be compensated by choosing an FI method that offers an additional degree of precision, such as LFI or EMFI, as performing FI by clock glitching might be too coarse-grained, which causes many unwanted side effects.

In contrast to the large difference in overall effectiveness between cores, a notable similarity is that in both experiments there does not seem to be a large difference between targeting different CSR access instructions. Since different CSR access instructions likely follow a very similar data path in the hardware, they are similarly affected by timing violations caused by clock glitches. Similarly, in both cases there exist some outliers, which could be due the small differences in the data path between instructions.

A limitation to be considered with the experiments is that a distinction is made between 'normal', 'reset' and 'success' behavior based on the observed output of the target, while there may have been faults that propagated and changed the state of the core, which were not observed. Since the target is not fully reset after a 'normal' execution in order to save time during the experiments, the core might have continued with a (hidden) corrupted state, possibly influencing the results of subsequent runs. To remove the possibility of this happening, one could opt to reset the target after every single FI attempt. However, in this study the inaccuracy is accepted and made justifiable by the fact that it saves several orders of magnitude in time, allowing for much more data to be collected, offsetting the inaccuracy per FI attempt.

7.3 Mitigation of Faults

The third subquestion focuses on the mitigation of FI attacks on TEEs for RISC-V. Some of them can be logically reasoned using available knowledge of each microarchitecture, while others have been discovered during the execution of the experiments and development of the methodology.

First, since a CSR write causes the pipeline to be flushed in both the E31 and the Ibex core (see section 5.2.1 and 5.3.2 respectively for more details), subsequent CSR writes to the same CSR are executed independently of each other. Meaning if one performs the same CSR write n times and each one has a probability $P_{success}$ of being successfully skipped using an injected fault, the probability of successfully attacking the software would be $P^n_{success}$. By simply introducing redundancy in the software, the probability of a successful attack can be greatly reduced. This would be quite a strong countermeasure that is generally applicable to RISC-V cores, since CSR writes will usually cause a pipeline flush or similar event in a pipelined RISC-V core, as a CSR write should have an immediate effect on the subsequently executing instructions. For example, if a memory access instruction is preceded by a CSR write that performs a PMP reconfiguration, the CSR write should be fully executed before the memory access instruction to prevent the memory access from being checked against an outdated PMP configuration.

Even though replicating CSR writing instructions is a great method for reducing the effectiveness of the attack scheme used in this study, possibly greatly reducing the success chance just by using a few redundant instructions, it is only applicable to attacks that target CSR writes. Although RISC-V CSRs are a prime target for attackers because of their influence over the behavior of the core, it might not be enough to only protect CSR writing operations from being targeted.

Other than using the inherent structure of RISC-V cores to mitigate CSR write-based attacks in software, there also exist more general hardware and software countermeasures that could be implemented, which are described in more detail in section 3.2. For example, methods such as dual core lockstep, which operates by comparing the output of a duplicated core against the delayed output of the real core, could be employed as a more overarching countermeasure against fault injections. Such methods have been shown to be quite effective and can even be enabled on the Ibex core. However, they often come at a significant cost in terms of hardware area usage and/or execution time, as they commonly introduce redundancy in either space, time, information or a combination of those. For example, dual core lockstep will always more than double the needed area, as the entire core is duplicated and a comparison mechanism is added.

Chapter 8

Conclusion

To conclude this work, it is confirmed that Trusted Execution Environments (TEE) on RISC-V have inherent commonalities that can be targeted using Fault Injection (FI) attacks. Any TEE that makes use of RISC-V's Physical Memory Protection (PMP) extension, which most notable RISC-V TEEs (such as Keystone [23]) do, needs to configure it appropriately using Control and Status Register (CSR) access instructions. When one of these instructions is skipped by means of an FI attack, the PMP configuration is corrupted, which may lead to a breach in the isolation that the TEE should provide between user applications.

Additionally, it was found that using different RISC-V cores heavily impacted the results obtained during the experiments; when assessing the vulnerability of a device with respect to FI attacks, it is important to consider its core's microarchitecture in addition to its ISA. While the FE310 target board with a hard E31 RISC-V core was shown to be vulnerable against clock glitching in the first experiment, the same cannot be said for the Ibex soft core used in the second experiment. Where the E31 showed successful attacks across a range of glitch shapes, the attacks on Ibex only showed normal or reset behavior. This behavior could be attributed to the Ibex's shorter pipeline, which limits the attacker's range of control over the injected fault, possibly causing too many unwanted side effects. However, to obtain a more conclusive answer as to which microarchitectural differences specifically contribute to a reduced susceptibility to FI attacks, a more comprehensive study should be conducted.

Finally, it can be argued that there is quite an effective way to mitigate the attack on a TEE as performed in the case study, namely by duplicating the CSR writing instruction that configures the PMP. Since it is a commonality between RISC-V cores that CSR writing operations are executed independently of each other, thus having an independent probability ($P_{success}$) of being successfully skipped or corrupted by an FI attempt, each duplicate instruction that is added reduces the probability of successfully skipping all of them by a factor of $P_{success}$. Furthermore, general fault tolerance techniques could be employed to cover a wider array of attack vectors. For example, dual core lockstep which is supported by certain configurations of the Ibex core. However, these techniques often introduce redundancy, which generally results in more hardware area usage and/or execution time.

8.1 Future Work

While the results seen in this work answer the research questions posed in the beginning, they also bring forth some speculation and follow-up questions, opening up avenues for future research. For example, while the experiments using the Ibex core did offer some insights into its vulnerability to clock glitching, a desired fault never occurred. It was concluded that this could be due to a lack of control over the injected fault due to the Ibex's shorter pipeline, prompting the need for a more precise FI method that compensates for the lack of precision. For example, EMFI, LFI or even XFI (see section 2.1.1) could be considered in future research for their additional precision at the expense of cost efficiency and/or ease of use.

Finally, although this work gives a clear indication that microarchitectural differences play a large role in a core's susceptibility to FI attacks, simply comparing 2 cores does not give sufficient insight to deduce which specific microarchitectural details contribute the most. In future research, a more comprehensive analysis could be performed by comparing a range of more similar cores, each with a small variation in its microarchitecture. Reducing the amount of differences between each core should help to narrow down which details are the most influential.

Bibliography

[1] Inc. Advanced Micro Devices. Amd secure encrypted virtualization (sev) - developer documentation. URL: https://www.amd.com/en/developer/sev.html.

[2] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. Proceedings of the IEEE, 94(2):370–382, 2006.

[3] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. Proceedings of the IEEE, 100(11):3056–3076, 2012.

[4] Arthur Beckers, Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. Design and implementation of a waveform-matching based triggering system. In Constructive Side-Channel Analysis and Secure Design: 7th International Workshop, COSADE 2016, Graz, Austria, April 14-15, 2016, Revised Selected Papers 7, pages 184–198. Springer, 2016.

[5] Daniel Binder, Edward C Smith, and AB Holman. Satellite anomalies from galactic cosmic rays. IEEE Transactions on Nuclear Science, 22(6):2675–2680, 1975.

[6] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In International conference on the theory and applications of cryptographic techniques, pages 37–51. Springer, 1997.

[7] Jakub Breier and Xiaolu Hou. How practical are fault injection attacks, really? IEEE Access, 10:113122–113130, 2022.

[8] Victor Costan. Intel sgx explained. IACR Cryptol, EPrint Arch, 2016.

[9] Rahul Dhobi, Sumit Gajjar, Dipen Parmar, and Tejas Vaghela. Secure firmware update over the air using trustzone. In 2019 Innovations in Power and Advanced Computing Technologies (i-PACT), volume 1, pages 1–4, 2019. doi:10.1109/i-PACT44901.2019.8959992.

[10] Mahmoud A Elmohr, Haohao Liao, and Catherine H Gebotys. Em fault injection on arm and risc-v. In 2020 21st International Symposium on Quality Electronic Design (ISQED), pages 206–212. IEEE, 2020.

[11] Clément Fanjas, Driss Aboulkassimi, Simon Pontié, and Jessy Clédière. Exploration of system-on-chip secure-boot vulnerability to fault-injection by side-channel analysis. In 2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pages 1–6, 2023. doi:10.1109/DFT59622.2023.10313346.

[12] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the PENGLAI enclave. In 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), pages 275–294. USENIX Association, July 2021. URL: https://www.usenix.org/conference/osdi21/presentation/feng.

[13] Fraktal. Fraktal - laser fault injection (lfi) rig. URL: https://github.com/fraktalcyber/lfi-rig.

[14] U.S. Government. U.s. government protection profile for separation kernels in environments requiring high robustness, version 1.03. URL: https://www.niap-ccevs.org/protectionprofiles/65.

[15] Donald H Habing. The use of lasers to simulate radiation-induced transients in semiconductor devices and circuits. IEEE Transactions on Nuclear Science, 12(5):91–100, 1965.

[16] Richard W Hamming. Error detecting and error correcting codes. The Bell system technical journal, 29(2):147–160, 1950.

[17] NewAE Technology Inc. Newae technology inc. URL: https://www.newae.com/.

[18] Henrik Karlsson. Openmz: a c implementation of the multizone api, 2020.

[19] Martin S Kelly and Keith Mayes. High precision laser fault injection using low-cost components. In 2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pages 219–228. IEEE, 2020.

[20] Koren. Reliability analysis of n-modular redundancy systems with intermittent and permanent faults. IEEE Transactions on Computers, 100(7):514–520, 1979.

[21] Raghavan Kumar, Philipp Jovanovic, and Ilia Polian. Precise fault-injections using voltage and temperature manipulation for differential cryptanalysis. In 2014 IEEE 20th International On-Line Testing Symposium (IOLTS), pages 43–48. IEEE, 2014.

[22] Johan Laurent, Vincent Beroulle, Christophe Deleuze, and Florian Pebay-Peyroula. Fault injection on hidden registers in a risc-v rocket processor and software countermeasures. In 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 252–255. IEEE, 2019.

[23] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In Proceedings of the Fifteenth European Conference on Computer Systems, pages 1–16, 2020.

[24] Arm Limited. Arm developer documentation - trustzone for aarch64. URL: https://developer.arm.com/documentation/102418/0101/Overview.

[25] Tung Lun Loo, Mohamad Khairi Ishak, and Khalid Ammar. Design and implementation of secure boot architecture on risc-v using fpga. Microprocessors and Microsystems, 101:104889, 2023. URL: https://www.sciencedirect.com/science/article/pii/S0141933123001321, doi:10.1016/j.micpro.2023.104889.

[26] lowRISC C.I.C. Documentation on instruction decode and execute pipeline stage of ibex risc-v core. URL: https://ibex-core.readthedocs.io/en/latest/03_reference/instruction_decode_execute.html.

[27] lowRISC C.I.C. Ibex demo system. URL: https://github.com/lowRISC/ibex-demo-system.

[28] lowRISC C.I.C. Ibex risc-v core. URL: https://github.com/lowRISC/ibex.

[29] Yifan Lu. Injecting software vulnerabilities with voltage glitching. arXiv preprint arXiv:1903.08102, 2019.

[30] Philippe Maurine, Karim Tobich, Thomas Ordas, and Pierre Yvan Liardet. Yet Another Fault Injection Technique : by Forward Body Biasing Injection. In YACC'2012: Yet Another Conference on Cryptography, Porquerolles Island, France, September 2012. URL: https://hal-lirmm.ccsd.cnrs.fr/lirmm-00762035.

[31] Shoei Nashimoto, Daisuke Suzuki, Rei Ueno, and Naofumi Homma. Bypassing isolated execution on risc-v using side-channel-assisted fault-injection and its countermeasure. IACR Transactions on Cryptographic Hardware and Embedded Systems, pages 28–68, 2022.

[32] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. Trustzone explained: Architectural features and use cases. In 2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC), pages 445–451. IEEE, 2016.

[33] Colin O'Flynn. Low-cost body biasing injection (bbi) attacks on wlcsp devices. In Smart Card Research and Advanced Applications: 19th International Conference, CARDIS 2020, Virtual Event, November 18–19, 2020, Revised Selected Papers 19, pages 166–180. Springer, 2021.

[34] Dmytro Petryk, Zoya Dyka, and Peter Langendoerfer. Optical fault injections: a setup comparison. RESCUE-Interdependent Challenges of Reliability, Security and Quality in Nanoelectronic Systems Design, 6, 2018.

[35] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pages 195–209, 2019.

[36] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not. In 2015 IEEE Trustcom/BigDataSE/Ispa, volume 1, pages 57–64. IEEE, 2015.

[37] Jörn-Marc Schmidt and Michael Hutter. Optical and em fault-attacks on crt-based rsa: Concrete results. na, 2007.

[38] Hex Five Security. Multizone® security tee for risc-v processors. URL: https://github.com/hex-five/multizone-sdk.

[39] Inc. SiFive. Sifive e31 manual v2p0. URL: https://static.dev.sifive.com/SiFive-E31-Manual-v2p0.pdf.

[40] Inc. SiFive. Sifive fe310-g002 datasheet v1p2. URL: https://starfivetech.com/uploads/fe310-g002-datasheet-v1p2.pdf.

[41] Sergei P Skorobogatov and Ross J Anderson. Optical fault induction attacks. In Cryptographic Hardware and Embedded Systems-CHES 2002: 4th International Workshop Redwood Shores, CA, USA, August 13–15, 2002 Revised Papers 4, pages 2–12. Springer, 2003.

[42] Nasr-Eddine Ouldei Tebina, Nacer-Eddine Zergainoh, and Paolo Maistri. X-ray fault injection: Reviewing defensive approaches from a security perspective. In 2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pages 1–4, 2022. doi:10.1109/DFT56152.2022.9962362.

[43] Nikolaus Theissing, Dominik Merli, Michael Smola, Frederic Stumpf, and Georg Sigl. Comprehensive analysis of software countermeasures against fault attacks. In 2013 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 404–409. IEEE, 2013.

[44] Jasper GJ Van Woudenberg, Marc F Witteman, and Federico Menarini. Practical optical fault injection on secure microcontrollers. In 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, pages 91–99. IEEE, 2011.

[45] Paulo RC Villa, Rodrigo Travessini, Roger C Goerl, Fabian L Vargas, and Eduardo A Bezerra. Fault tolerant soft-core processor architecture based on temporal redundancy. Journal of Electronic Testing, 35:9–27, 2019.

[46] Andrew Waterman, Krste Asanović, and John Hauser. The RISC-V Instruction Set Manual. Volume II: Privileged Architecture. RISC-V International, December 2021. Document Version 20211203. URL: https://drive.google.com/file/d/1EMip5dZlnypTk7pt4WWUKmtjUKTOkBqh/view.

[47] Christian Werling, Niclas Kühnapfel, Hans Niklas Jacob, and Oleg Drokin. Jailbreaking an electric vehicle in 2023 or what it means to hotwire tesla's x86-based seat heater, 2023. Slideshow presented at Blackhat 2023.

[48] Mario Werner, Robert Schilling, Thomas Unterluggauer, and Stefan Mangard. Protecting risc-v processors against physical attacks. In 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1136–1141. IEEE, 2019.

[49] Marc Witteman. Fault mitigation patterns. Riscure Whitepaper.

[50] Yongwang Zhao, Zhibin Yang, and Dianfu Ma. A survey on formal specification and verification of separation kernels. Frontiers of Computer Science, 11:585–607, 2017.