Embedded Systems Security: On EM Fault Injection on RISC-V and BR/TBR PUF Design on FPGA

by

Mahmoud A. Elmohr

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2020

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

With the increased usage of embedded computers in modern life and the rapid growth of the Internet of Things (IoT), embedded systems security has become a real concern. Especially with safety-critical systems or devices that communicate sensitive data, security becomes a critical issue. Embedded computers more than others are vulnerable to hardware attacks that target the chips themselves to extract the cryptographic keys, compromise their security, or counterfeit them.

In this thesis, embedded security is studied through two different areas. The first is the study of hardware attacks by investigating Electro Magnetic Fault Injection (EMFI) on a RISC-V processor. And the second is the study of the countermeasures against counterfeiting and key extraction by investigating the implementation of the Bistable Ring Physical Unclonable Function (BR-PUF) and its variant the TBR-PUF on FPGA.

The experiments on a 320 MHz five-stage pipeline RISC-V core showed that with the increase of frequency and the decrease of supplied voltage, the processor becomes more susceptible to EMFI. Analysis of the effect of EMFI on different types of instructions including arithmetic and logic operations, memory operations, and flow control operations showed different types of faults including instruction skips, instructions corruption, faulted branches, and exception faults with variant probabilities. More interestingly and for the first time, multiple consecutive instructions (up to six instructions) were empirically shown to be faulted at once, which can be very devastating, compromising the effect of software countermeasures such as instruction duplication or triplication.

This research also studies the hardware implementation of the BR and TBR PUFs on a Spartan-6 FPGA. A comparative study on both the automatic and manual placement implementation approaches on FPGA is presented. With the use of the settling time as a randomization source for the automatic placement, this approach showed a potential to generate PUFs with good characteristics through multiple trials. The automatic placement approach was successful in generating 4-input XOR BR and TBR PUFs with almost ideal characteristics. Moreover, optimizations on the architectural and layout levels were performed on the BR and TBR PUFs to reduce their footprint on FPGA.

This research aims to advance the understanding of the EMFI effect on processors, so that countermeasures may be designed for future secure processors. Additionally, this research helps to advance the understanding of how best to design improved BR and TBR PUFs for key protection in future secure devices.

Acknowledgements

First and foremost, I would like to express my sincere appreciation to my supervisor, Professor Catherine Gebotys. Without her guidance and support, this thesis would not have been possible.

Many thanks to Professor Nachiket Kapre for his insights and advice at the early stages of the PUF design.

I also sincerely appreciate the time and effort spent by the thesis review committee, Professor Hiren Patel and Professor Kshirasagar Naik.

Thanks are extended to my labmates: Bahareh Sadrabadi, Haohao Liao, Karim Amin, Mahmoud Khalafallah, and Mustafa Faraj for the encouragement and their friendship.

Special recognition to Mahmoud Khalafallah for the fruitful collaboration on the PUF research and to Haohao Liao for laying the groundwork for the EMFI research. Their valuable discussions and the experience they shared contributed significantly to the thesis progress.

Finally, I would like to express my gratitude to my parents Aly Elmohr and Nagwa Elbedawy, and my sister Alaa Elmohr for their endless love and support.

Dedication

This thesis is dedicated to my parents, for they have always believed in me, supported me, and the reason I have reached this point in my life.

Table of Contents

List of Tables

List of Figures

Glossary

CNC machine Computer Numerical Control Machine: a motorized maneuverable tool which is controlled by a computer 27, 29, 30, 34

JTAG Joint Test Action Group: an industry standard for verifying designs and testing printed circuit boards after manufacture 30, 35

RISC-V An open source Instruction Set Architecture (ISA) based on the Reduced Instruction Set Computer (RISC) principal 2, 4, 13, 30, 33, 34, 46, 52, 79, 81, 83, 84

SPICE Simulation Program with Integrated Circuit Emphasis: a general-purpose, open-source analog electronic circuit simulator used in integrated circuit design to predict circuit behavior 22

VDD Voltage Drain Drain: refers to the positive operating voltage of a field effect semiconductor device 27

Abbreviations

AES Advanced Encryption Standard 1, 2, 11, 12, 82

ALU Arithmetic Logic Unit 52

ANN Artificial Neural Network 23, 24

ASIC Application Specific Integrated Circuit 22, 23

BHT Branch History Table 33

BR Bistable Ring 3–5, 14, 15, 18–20, 22–26, 62, 64, 67–70, 73–77, 79, 80, 83, 84

BSPs Board Support Packages 30

BTB Branch Target Buffer 33

C.NOP Compressed No Operation 41, 44–47, 52, 55, 57, 59, 61

CAD Computer Aided Design 67, 73

CRPs Challenge-Response Pairs 15–17, 20–23, 25, 26, 65, 75, 77, 83

DTIM Data Tightly Integrated Memory 33

ECC Elliptic Curve Cryptography 2, 82

EEPROM Electrically Erasable Programmable Read-Only Memory 6, 14

EM Electro-Magnetic 2, 4, 7, 9–12, 27, 28, 30, 31, 34–36, 38, 41, 43, 45–47, 49, 51, 52, 79, 81–83

EMFI Electro-Magnetic Fault Injection 2, 4–7, 9–13, 27, 38, 39, 49, 52, 79–84

FIA Fault Injection Attack 2, 3, 13

FPGA Field-Programmable Gate Array 3–5, 11, 19, 22, 23, 25, 62–64, 67–69, 71, 73, 76, 79, 80, 83, 84

FSM Finite State Machine 63, 64

GPIO General Purpose Input Output 32, 43

GUI Graphical User Interface 30

HDL Hardware Description Language 67, 68, 83

IC Integrated Circuit 3, 14

IO Input Output 32, 81

IoT Internet of Things 1

ISA Instruction Set Architecture 2

ITIM Instruction Tightly Integrated Memory 33, 41–43

LFSR Linear-Feedback Shift Register 63

LR Linear Regression 20

LUT Look Up Table 62, 67, 68, 71, 73, 76, 80

MPSoC Multi-Processor System on Chip 3

PAC Probably Approximately Correct 24

PC Program Counter 58

PC Personal Computer 27, 29, 30, 63–65

PDA Personal Digital Assistant 1

PEM Photon Emission Microscopy 6

PLL Phase-Locked Loop 27, 32

POK Physically Obfuscated Key 14

PUF Physical Unclonable Function 3–5, 14–26, 62–70, 73–77, 79, 80, 83, 84

QFN Quad Flat No-leads 32

QSPI Quad Serial Peripheral Interface 32

RAM Random Access Memory 12, 14

RAS Return Address Stack 33

RO Ring Oscillator 14–16, 18

RSA Rivest–Shamir–Adleman 2, 82

RTL Register Transfer Level 13, 80, 84

SCA Side-Channel Attack 2, 3

SDK Software Development Kit 30

SHA-3 Secure Hash Algorithm 3 2

SoC System on Chip 3, 32

SPI Serial Peripheral Interface 32, 33

SRAM Static Random Access Memory 6, 14, 15, 18

SVM Support Vector Machine 20, 25, 26, 73, 75–77

TBR Twisted Bistable Ring 3–5, 14, 15, 19, 23–26, 62, 64, 67, 68, 73, 76, 77, 79, 80, 83, 84

UART Universal Asynchronous Receiver-Transmitter 63

USB Universal Serial Bus 27, 29, 30, 32, 63

Chapter 1

Introduction

This chapter presents the motivation behind this research in Section 1.1, states the goals of the thesis in Section 1.2, and finally goes through the thesis organization and gives a summary of each chapter in Section 1.3.

1.1 Motivation

From cars to digital cameras, MP3 players, digital watches, cell phones, pacemakers, and traffic light controllers, embedded computers increasingly saturate our lives. Many modern embedded systems such as Personal Digital Assistant (PDA), sensors, routers, and smart cards handle and communicate sensitive data, which makes their security a serious issue. Also, embedded devices are used in safety-critical systems such as automotive systems and medical care devices, making it very crucial to secure such systems [50]. Especially with the growth of the Internet of Things (IoT), more of the embedded devices are connected to the internet which imposes more vulnerabilities and scales up the number of possible attack vectors [37].

Computer security has been extensively researched in terms of goals and solutions. The main security goals could be outlined as confidentiality, data integrity, authentication and availability [77]. Cryptographic primitives such as symmetric-key ciphers, public-key ciphers, and one-way hash functions were introduced to help achieve the aforementioned security goals. Furthermore, protocols relying on combinations of these primitives can provide end-to-end security if designed carefully. With extensive research on logical attacks and cryptanalysis, standards and cryptosystems such as Advanced Encryption Standard

(AES) [68], Rivest–Shamir–Adleman (RSA) [51], Elliptic Curve Cryptography (ECC) [44] and Secure Hash Algorithm 3 (SHA-3) [19] as well as best practices for protocols were provided to make such logical attacks almost infeasible.

However, for embedded systems, more concerns arise due to their limited resources and the unprotected environments they operate in [50]. Thus, embedded systems are not only susceptible to software logical attacks. Embedded devices are more susceptible to physical attacks: a different kind of attack that operates on the physical hardware, exploiting the system implementation to breach its security. Such attacks can compromise the aforementioned standards that are expected to be secure against logical attacks. Examples can be found in [60], [23] and [47] attacking AES, RSA and ECC respectively.

Physical attacks can be passive, referred to as a Side-Channel Attack (SCA), where an attacker exploits the unintentional leakage from the device such as timing information, power consumption, heat radiation, Electro-Magnetic (EM) emissions or even sound. Passive SCA can be used to learn about the computations performed on the device while running the target cryptographic algorithm, and eventually may be used to recover the cryptographic keys [1]. Physical attacks can also be active as in Fault Injection Attack (FIA), where an attacker tries to influence the device to make a computational error. Such computational errors can lead to bypassing some security checks or creating new side-channels [1], which makes FIA more dangerous and powerful than SCA.

FIA can be launched using different techniques; Some are simple and inexpensive such as clock glitching and voltage glitching. These FIAs might be capable of skipping instructions or corrupting them, resulting in loading or storing faulty data in corrupted addresses [6]. However, typically the glitching effect is global, thus an attacker may not be able to address specific registers or specific areas of the chip. Also, access to the power line or clock line needs to be provided. Some other FIA can be complex and provide a local effect such as micro-probing, optical fault injection and Electro-Magnetic Fault Injection (EMFI). While micro-probing and optical fault injection can be very precise and powerful, they require the decapsulation of the chip package to gain direct access to the silicon die, which is a sophisticated process with the risk of damaging the chip [29]. The fact that EMFI does not require package decapsulation nor requires access to any pins or lines of the chip makes it a very powerful FIA technique. For that reason, this research focuses on EMFI attacks on a RISC-V microprocessor. The reasoning behind choosing RISC-V as a target is being a new open-source Instruction Set Architecture (ISA), which is gaining a lot of interest in academia and industry recently, and it has never been studied for EMFI attacks as per our knowledge.

Physical attacks can be very powerful. They can enable an attacker to skip authentication routines using FIA, and recover encryption keys with the help of SCA. But what is more devastating is that in some attacks, an attacker can even directly readout cryptographic keys stored in memories [69], which is a huge security breach. To face this threat, a new cryptographic primitive referred to as a Physical Unclonable Function (PUF) was introduced recently to overcome the key storage problem [27]. With the use of this primitive, instead of storing keys in memory, keys can be derived using physical one-way functions.

A PUF briefly is a physical one-way function that is practically infeasible to replicate. Two instances of the same PUF are expected to yield independently random responses (outputs) to the same challenges (inputs). Silicon PUFs, which are the most widely proposed architectures of PUFs, exploit the slight intrinsic variations during the Integrated Circuit (IC) fabrication as their source of randomness. Thus, they are practically infeasible to clone even with the knowledge of the architecture [9].

The most important advantage of silicon PUFs is that they provide an easy to realize PUF implementation on silicon chips, eliminating overheads required to interface the PUF to the chip. This helped PUFs to gain the interest not only of researchers but also industry. PUFs are now provided by many companies such as Intrinsic ID [32] and eMemory [22], and incorporated in several hardware platforms including Microsemi [43] SmartFusion2 System on Chip (SoC) Field-Programmable Gate Array (FPGA), NXP [48] SmartMX2 P60 family of secure controllers, Intel [31] Stratix 10 SoC FPGA and Xilinx [73] Zynq Ultrascale+ Multi-Processor System on Chip (MPSoC).

Generally, there are two main types of silicon PUFs: delay-based PUFs which depend on the delay measurements of a signal traversing two long symmetric paths controlled by the challenge bits, and memory-based PUFs which leverage the bi-stability nature of closed-loop circuits to generate responses [42]. Delay-based PUFs have the advantage of the large challenge-response space, while it suffers from the vulnerability against modeling attacks. However, memory-based PUFs are resilient against modeling attacks but lack the large challenge-response space. The Bistable Ring (BR) PUF introduced in [13] tries to get the best of both worlds by merging both delay-based PUFs and memory-based PUFs approaches to create a stronger PUF with a large challenge-response space, yet, resilient to modeling attacks. This newly introduced architecture alongside with its variant, the Twisted Bistable Ring (TBR) PUF [57] are very promising due to their characteristics. Hence, the motivation for this thesis to also investigate their application approaches on FPGA and their associated challenges.

1.2 Research Goals

In this masters thesis, two different research projects are presented: the first discusses EM fault injection on a RISC-V processor, and the other investigates the implementation approaches of BR and TBR PUFs on FPGA.

- The research goals of the EMFI part are outlined as follows:

    1. Developing a deeper understanding of the EMFI attacks and their capabilities on the state of the art RISC-V processor.
    2. Examining the conditions under which the processor is more susceptible to faults.
    3. Exploring the possibility of faulting multiple instructions at once.
    4. Analyzing the fault injection effect on different types of instructions (logical, arithmetic, memory operations, branches, and jumps).

- For the PUF part, the research goals are summarized below:

    1. Investigating the BR and TBR implementations on FPGA.
    2. Exploring different FPGA layout approaches such as manual placement and automatic placement and their effect on PUFs characteristics.
    3. Optimizing both PUFs on architecture level and layout level to reduce utilized area.
    4. Implementing 4-input XOR BR and TBR PUFs with good characteristics to serve as a base for launching more advanced modelling attacks, since the 4-input XOR BR PUF was not successfully modeled yet according to literature.

1.3 Thesis Organization

The rest of this thesis is organized as follows:

Chapter 2 provides an introduction to physical attacks and their types, it discusses fault injection techniques, with more emphasis on EMFI. Finally, it summarizes the previous research conducted on EMFI.

In **Chapter 3**, the PUF principals and characteristics are explained. The different types of PUF are briefly illustrated with more details on the BR PUF family. Also, a literature review is provided summarizing BR and TBR PUF related work.

Chapter 4 describes the EMFI experiments setup. It gives details on the hardware components and the software components used, as well as illustrates the target development board and its underlying processor.

In **Chapter 5** the EMFI experiments and their results are explained in detail. The chapter includes the different experiments, alongside with the codes used, as well as the results of each experiment.

Chapter 6 illustrates the BR and TBR implementations on FPGA and discusses their characteristics. The chapter describes the overall system architecture, and the ecosystem designed to accommodate the PUFs. It also depicts the metrics used to evaluate the PUFs characteristics. Details on the BR and TBR implementations on FPGA, the layout approaches and the optimizations performed to reduce the PUFs size are also provided.

Finally **Chapter 7** concludes the paper by summarizing its experiments and results, listing the main contributions and providing further research directions.

Since the thesis is compromised of two different research works, road maps for readers who are interested in only one part are presented as follows:

- For the EMFI part: Chapter 2 provides the background and summarizes the literature review. Then Chapters 4 and 5 illustrate the EMFI experiments setup and results respectively. And finally Chapter 7 provides the conclusion for both parts.

- For the PUF part: Chapter 3 presents the background and lists the previous research. Then Chapter 6 details the PUF implementations on FPGA and their characteristics. And similarly Chapter 7 concludes both parts.

Chapter 2

EMFI Background and Related Work

In this chapter an introduction to physical attacks and their types is provided in Section 2.1. The chapter also explains fault injection attacks and their techniques in Section 2.2 with more emphasis on EMFI in Section 2.3. Finally, the related work is listed in Section 2.4.

2.1 Physical Attacks

Physical attacks can be classified as invasive, semi-invasive and non-invasive according to their degree of physical penetration and access to the chip [4].

- **Invasive attacks** involve not only decapsulating the package of the chip but also getting physical access to the chip internals and even more sometimes include altering the chip's physical structure. An example of this type of attack is micro-probing attacks, which use microscopic needles onto the internal wiring of a chip; enabling it to inject faults or readout data buses revealing memory content [36].

- **Semi-invasive attacks** require decapsulating the chip but do not physically access the die neither modifies the circuitry. An example of this type is Photon Emission Microscopy (PEM) which uses photon emissions associated with the switching of transistors to observe the data processed inside semiconductor chips. With this ability, data stored in Static Random Access Memory (SRAM), Electrically Erasable Programmable Read-Only Memory (EEPROM) and Flash memories can be extracted [65].

- **Non-invasive attacks** as the name indicates, do not physically alter the chip nor its packaging. EMFI could be considered non-invasive if the packaging is not decapsulated. EMFI utilizes EM pulses directed to the chip inducing current that changes the processor's transistors states. These changes can cause instructions skips or corruptions leading to skipping authentication routines or recovering keys with some additional analysis [18].

Another classification is based on the attacker's role in the execution of the cryptographic modules into passive and active attacks.

- **Passive (side-channel) attacks** in which the attacker only analyzes the physical leakage from the executing device while running the cryptographic module, to learn about the computations performed and recover cryptographic keys. The source of leakage could be the device's power consumption, EM emission, or time variations [1].
- **Active (fault injection) attacks** in which the attacker actively manipulates the device running the cryptographic module, causing computational errors that can be used to skip authentication routines or to create other side-channel leakages. The techniques used to cause faults include clock glitches, laser beams, and EMFI [6].

2.2 Relevant Fault Injection Techniques

As explained earlier, fault injection is the process of intentionally influencing an electronic device's behavior to deviate it from its proper functionality [29]. In fault injection attacks, these intentional faults are used to breach the system's security with some knowledge of the system being attacked.

Different techniques can be used to achieve fault injection. In this section the most common techniques are briefly explained including clock glitching, voltage glitching, microprobing and optical fault injection, while EMFI is explained in detail in Section 2.3.

2.2.1 Clock glitching

Clock glitching relies on the modification on the clock signal of the target device for a fixed period to cause faults. In this method, the attacker increases the clock frequency to fall below the critical path of the circuit, causing incorrect data writes to registers [29].

Different kinds of faults can be injected such as instructions being corrupted or skipped, corrupted data being loaded to memory or registers or corrupted addresses being used for memory operations or jump instructions [6]. The clock glitch effect depends on which registers are affected by that critical path violation.

Clock glitching is one of the easiest to deploy techniques, but also one of the easiest to countermeasure by adding extra circuitry to detect and control the clock behavior.

2.2.2 Voltage glitching

Voltage glitching similarly relies on increasing or decreasing the voltage supplied to the device above or below the threshold for a fixed period of time to cause faults [7].

This technique can cause behavioral changes to the circuit as registers might fail to hold their values, or when voltage is decreased logic levels might not be raised to the correct values and get interpreted wrongly [1]. Also, short variations in supplied voltage known as spikes can cause other types of faults affecting memory and code flow [79].

2.2.3 Micro-probing

In the micro-probing method, tiny needles called probes are placed on internal signal lines on the die. The probes can be used to read and observe the internal signals which might be used as a side-channel, or even data buses can be readout revealing memory content [36]. Micro-probing can also be used for fault injection, as the probes can be used to overwrite the internal signals permanently or at a specific time during execution [66].

One disadvantage of micro-probing is being invasive, meaning that the chip has to be decapsulated and often further modified using focused ion beam systems to access the internal signals. Also, countermeasures detecting the probes are promising and effective.

2.2.4 Optical fault injection

Optical fault injection exploits the fact that transistors are vulnerable to photons. With the use of strong laser beams, an attacker can switch on or off transistors [1]. Since laser beams are very narrow compared to standard light, precise targeting to specific transistors is feasible [70].

The precision provided by the optical fault injection attack makes it very powerful, allowing the attacker to manipulate the instruction itself by flipping specific bits in the instruction register during the fetch stage and change the target instruction into another. However, the disadvantages include the possible requirement of chip decapsulation as well as the higher risk of damaging the chip if the laser energy is high [1].

2.3 Electro Magnetic Fault Injection

EMFI is the use of electromagnetic induction to alter the functionality of the device. The attacker uses an EM probe to generate a magnetic flux that is responsible for inducing a current in the target device, which alters the device operation. This method of fault injection is more localized than the glitching techniques and less invasive than optical or micro-probing approaches since often decapsulation is not required.

According to the Maxwell-Faraday equation, varying a current in a ring-shaped conductor generates a magnetic field and vice versa. The intensity of the magnetic field B in a ring-shaped conductor with radius R and current I in vacuum is calculated according to Equation 2.1. Where μ_{o} is the magnetic constant and μ_{r} is the magnetic permeability of the material that the magnetic field is passing through.

$$B = \frac{\mu_{\mathrm{o}}\mu_{\mathrm{r}} I}{2R} \tag{2.1}$$

Which could be formulated for a one loop coil as in Equation 2.2 according to [29]. Where b is the radius of the coil and z is the distance from the plane of the coil.

$$B = \frac{\mu_{\mathrm{r}} I b^2}{2(b^2 + z^2)^{\frac{3}{2}}} \tag{2.2}$$

Then the magnetic flux across a surface plane with an area A can be calculated by Equation 2.3. Where θ is the angle between the surface plane and the coil plane.

$$\phi_{\mathrm{B}} = BAcos(\theta) = \frac{\mu_{\mathrm{r}} I b^2 Acos(\theta)}{2(b^2 + z^2)^{\frac{3}{2}}} \tag{2.3}$$

Similarly, according to the Maxwell-Faraday equation, varying a magnetic field generates a potential across a loop conductor and induces current that depends on the time derivative of the magnetic flux as in Equation 2.4.

$$\oint E.dl = -\frac{d}{dt}\iint B.dS \tag{2.4}$$

In summary, to apply EMFI a variant current should be applied using an EM probe to generate magnetic flux. The variation in current could be harmonic as a sinusoidal signal or a transient pulse. The magnetic flux generated by the probe tip would induce a current in the target device to influence its operation.

To maximize the current induced in the device, the time derivative of the magnetic flux should be maximized. Thus, the shorter the rise time of the EM pulse the higher the derivative, hence the higher the induced current. Also, to maximize the magnetic flux itself, which in turn affects its time derivative and the induced current, the parameters in Equation 2.2 should be considered by:

- $\boldsymbol{\mu}_{\mathrm{r}}$: Making the probe tip from a material with a high magnetic permeability.
- **I:** Increasing the current of the pulse.
- $\boldsymbol{\theta}$: Placing the probe tip to be perpendicular to the target chip to make $\theta = 0$, thus $cos(\theta) = 1$.
- **z:** reducing the distance between the probe tip and the target chip.

2.4 EMFI Previous Research

Since the introduction of the EMFI concept, many studies on EMFI have been reported in the literature studying its behavior on different targets and exploiting EMFI for practical attacks. In [17] Dehbaoui et al. applied EMFI on AES implementations on both 8-bit AVR Atmega-128 and a Xilinx Spartan-3 FPGA. Experiments on the AVR microcontroller showed faults on the various AES rounds, some of the faults depend on the cipher-text and some are constant. The occurrence rate of the faults was found to be positively correlated with the EM pulse amplitude, reaching a success rate of 100% for a pulse amplitude of 100 V. It was also observed that a negative spike in the power supplied to the target occurs upon applying the EM pulse. However, the authors related the faults behavior of their experiments to a similar behavior observed in [6] on the same target responding to clock glitching. Thus, it was concluded that the EMFI caused timing violations similar to the clock glitching effect.

The experiments on the hardware implementation of AES on FPGA showed variant faults depending on the location of the probe tip over the chip. Which shows a local effect in contrast to the global effect caused by clock glitching. The authors explained that local behavior by assuming the possibility of having multiple sub-critical paths that can be affected depending on the probe location. To verify their hypothesis, a countermeasure was implemented on the FPGA implementation to monitor the data path delay and activates an alarm signal when timing constraints are violated. However, according to the experiments performed on different locations on the chip for multiple rounds, only 10% of the faults injected was accompanied by the trigger of the alarm signal. Moreover, more than 95% of the triggered alarm signals were false alarms without any faults injected.

In [49] Ordas et al. discussed the effectiveness of the delay fault model introduced in [17], and extended it with the sampling fault model. The experiments conducted on both Xilinx Spartan-3E FPGA and ARM cortex M4 microcontroller showed that to induce a fault, there are two EM power thresholds: $\mathrm{V}_{\mathrm{Low}}$ and $\mathrm{V}_{\mathrm{High}}$. When an EM pulse above $\mathrm{V}_{\mathrm{High}}$ is produced at any time, bit-set or bit-reset faults are injected, which complies with the previous delay fault model. More interestingly, if the EM pulse is below $\mathrm{V}_{\mathrm{High}}$ but above $\mathrm{V}_{\mathrm{Low}}$, faults can be injected only if the pulse is produced just before the occurrence of the rising clock edge or more precisely during the sampling window of the registers. The increased susceptibility of the chips for EMFI near the rising edge supports the sampling fault model, as if the delay fault model was correct, faults would have been injected regardless of the timing of the EM pulse.

In [40], Liao et al. proposed the charge-based fault model. The experiments conducted on a PIC16F687 microcontroller complied with the sampling fault model introduced in [49] in the sense that the microcontroller is susceptible before the rising edge of the clock. However, when the clock frequency was reduced, the fault injection was not possible. Moreover, with these lower clock frequencies, faults could be injected when the supplied voltage to the microcontroller is decreased below the nominal value.

The explanation provided by the authors is that considering a node charging from 0 to 1 logic levels, with slow clock frequency a maximum amount of charge is built up on the capacitors at that node. In that case, the EM pulse would not be able to reduce a sufficient amount of charge on that node to change its level to 0. However, with faster clock frequencies, the charge would be accumulated for a short time resulting in less charge being accumulated, which can be affected by the EM pulse. Similarly, with reduced voltage supplied to the target, the capacitance would charge slower resulting in a less accumulated charge during the clock period, making it possible for the EM pulse to affect that charge and change it to 0.

The paper also discussed the effect of the EMFI on the PIC16F687 microcontroller and the instruction replacements with their corresponding probabilities. Also, an attack on AES to retrieve its key was demonstrated by exploiting the fault on the XORWF instruction which is used to XOR the key byte with the AES state.

Cui et al. in [16] discussed the use of the second-order EMFI attacks. Unlike first-order EMFI attacks which are directed towards the target to introduce deterministic faults, second-order EMFI attacks are directed towards other components interacting with the target, such as memories and system interconnects.

The authors applied a second-order EMFI on an ARM processor to defeat a TrustZone-based secure boot implementation. The attack relies on corrupting the data in the Random Access Memory (RAM) which contains the secure boot code, causing the secure boot to enter a fault condition that starts executing the debug command-line interface. Accessing the debug command line allows the attacker to modify the TrustZone memory and execute an arbitrary code bypassing the secure boot.

In summary, different studies on EMFI have been reported in the literature. Different fault models have been introduced such as the delay fault model in [17], the sampling fault model in [49], and the charge-based model in [40]. Moreover, other researchers have applied EMFI in attacking real applications such as secure boot [16], and AES [40].

The EMFI attacks were proposed targeting different microcontroller architectures including PIC [40], AVR [17] and ARM [49]. In these studies injected faults involved single instruction skips, single instruction corruption or caused exceptions. EMFI has never been studied on the RISC-V architecture in the literature although some research papers established emulation framework for FIA on RISC-V in [20], launched an FIA simulation on the Register Transfer Level (RTL) of a RISC-V core in [39], and proposed countermeasures for the RISC-V architecture to resist possible FIA in [72]. However, these studies are based on simulations with no empirical results, and are not specific to EMFI.

Chapter 3

PUF Background and Related Work

This chapter provides an introduction to silicon PUFs and classifies their types in Section 3.1. It also goes through the different PUFs architectures in Section 3.2, with more emphasis on BR and TBR PUFs in Section 3.3. The XOR PUF is illustrated in Section 3.4 followed by a discussion on the characteristics required for a strong PUF in Section 3.5. And finally, the previous research is discussed in Section 3.6.

3.1 Physical Unclonable Functions

PUFs have been introduced in the last 15 years as promising primitives acting as die-specific random functions. Such primitives can be used for authentication and cryptographic key generation without the need for storing keys in secure EEPROM or battery-packed RAM. Since the introduction of silicon PUFs in [27], many architectures have been introduced in the literature, providing different techniques to extract randomness from the intrinsic variations during the IC fabrication [9]. These architectures can be classified into two main groups according to their challenge-response space into strong and weak PUFs [55].

- **Weak PUFs** have few fixed challenges, sometimes just a single challenge. This type of PUFs can also be called a Physically Obfuscated Key (POK) [28], since they are mainly used for internal key derivation. Although POKs or weak PUFs are harder to read out using invasive attacks compared to digital storage in non volatile memory, the secret keys are still susceptible to side channel attacks as in any physical cryptosystem [67]. Examples for weak PUFs would be: the SRAM PUF [30] and the Ring Oscillator (RO) PUF [69].

- **Strong PUFs** have a very large challenge-response space. The challenge-response space should be too large to readout all the corresponding Challenge-Response Pairs (CRPs) in a feasible time. The CRPs ideally should be independent and complex in a way that no adversary can drive unknown CRPs from a set of known ones. Thanks to their large CRPs space, strong PUFs can be used for authentication using challenge-response protocols.

  A basic procedure would work as follows: a trusted third party reads out a large number of CRPs for each device and save them to a database. When authentication is needed, the verifier sends challenges to the device and compares the responses received to the ones stored in the database [57].

  Strong PUFs however, are susceptible to machine learning modelling attacks, which makes constructing a practical strong PUF very difficult and considered to be still an open problem [53]. Examples for strong PUFs would be: the arbiter PUF [41] and the BR PUF [13].

3.2 Relevant PUF Architectures

Different architectures have been introduced to implement PUFs. In this section, the most popular architectures are explained such as the SRAM PUF, the RO PUF and the arbiter PUF, while the BR and TBR PUFs are explained in detail in Section 3.3

3.2.1 SRAM PUF

The SRAM PUF [30] utilizes the fact that an SRAM cell upon power-up would be randomly initialized to 0 or 1 without any write operation. As shown in Figure 3.1, an SRAM cell can be viewed as two cross-coupled inverters. When no power is applied, the output of both inverters Q and $\overline{Q}$ is 0. When power is applied, both inverters try to pull up their outputs to 1. Which puts the cell in a metastable state since the inverters are cross-coupled. But practically, due to mismatches between transistors, one inverter would have a stronger pull-up than the other and the cell will tend to stabilize storing 0 or 1 depending on the process variation per each cell. Since the random process variations at the time of manufacturing are random, and hence is cell-specific, SRAM cells can be used as PUFs.

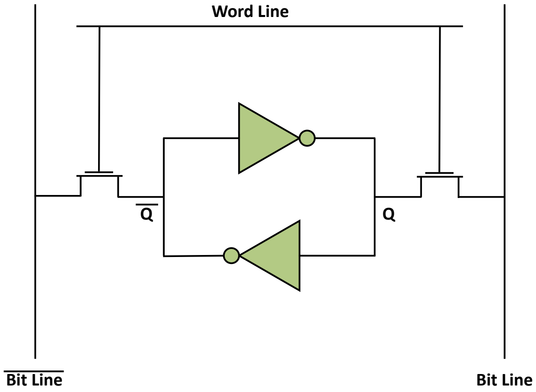


Figure 3.1: SRAM cell

3.2.2 RO PUF

The RO PUF [69] is based on measuring the frequencies of digital oscillator circuits. The architecture depicted in Figure 3.2 consists of multiple instances of a ring oscillator, each is built using an odd number of inverters in a ring. The uncontrollable process variation causes inverters to have slightly different delays. This variation in gates delays in each ring oscillator causes each to have a different frequency slightly deviating from the design value. Two frequency counters, as well as a comparator, are used to produce the response of 0 or 1 depending on which ring oscillator has a higher frequency. To accommodate challenges, two multiplexers are introduced to feed the frequency counters the outputs of two instances out of the multiple instances exist. The RO PUF's challenge-response space is not considered large though, as from N instance of ring oscillators only $\binom{N}{2}$ CRPs can be produced, which is equivalent to $\frac{N(N-1)}{2}$. However, many of these pair-wise comparisons are redundant, resulting in only a maximum of independent $\log_2 N!$ CRPs available for N instances, which is is not large, making RO PUF to still be considered a weak PUF [53].

3.2.3 Arbiter PUF

The arbiter PUF [41] is one of the first introduced architectures for strong PUFs. The idea behind the arbiter PUF is basically to establish a race between two symmetric digital paths. A step signal is applied to both paths and an arbiter circuit at the end of both paths determines which path is faster and outputs the response bit accordingly. Again the uncontrollable process variation in silicon would make the delay of both paths slightly

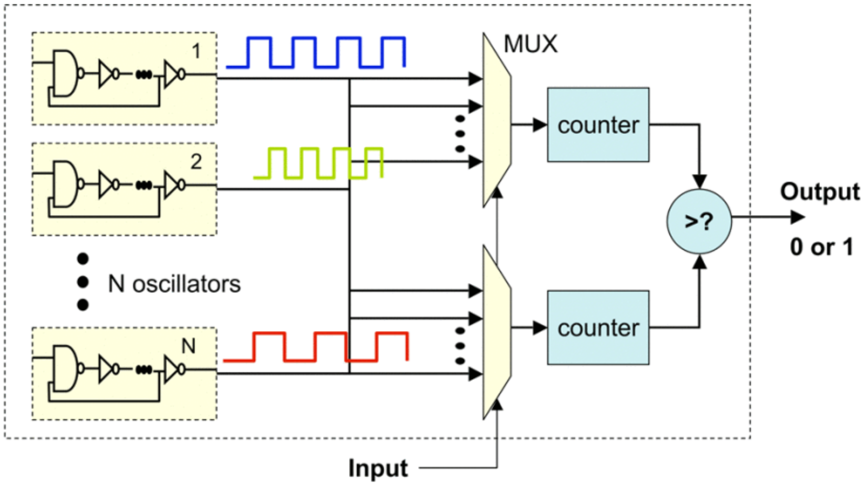


Figure 3.2: RO PUF architecture [69]

different, favoring one path on another. To incorporate challenges, the architecture in Figure 3.3 is used by implementing the two paths as a chain of switch blocks controlled by the challenge bits. The switch block, which could be realized using two multiplexers, is basically a circuit that connects two input signals to its two outputs, by either crossing the inputs or traversing them straight. This architecture has the advantage of having exponential CRPs. For N switch blocks, 2^N CRPs can be produced. However, as delays of each stage is independent and accumulative, the arbiter PUF could be considered a linear structure, which makes it easily modeled [53].

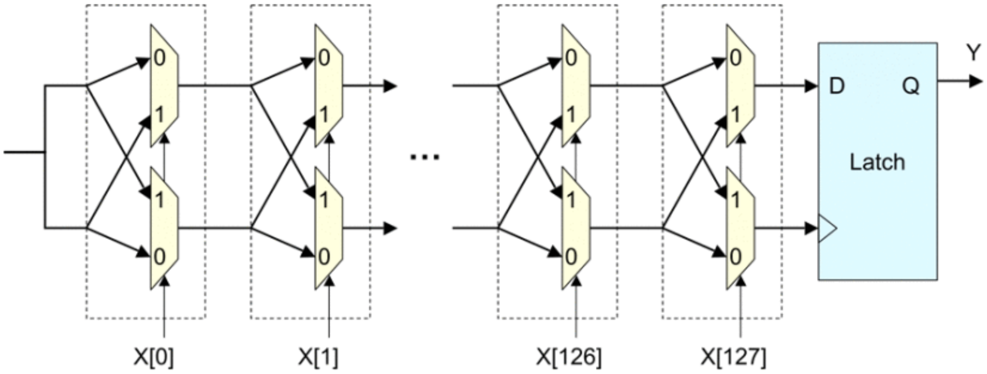


Figure 3.3: Arbiter PUF architecture with 128 stages [69]

3.3 BR and TBR PUFs

The BR PUF combines the elements of the RO PUF and the SRAM PUF with an exponential challenge response space similar to the Arbiter PUF. Its conceptual idea is based on the fact that a ring of even number of inverters eventually converges to only two possible stable states. To illustrate more, consider the ring of eight inverters in Figure 3.4. If the inverters are initialized to all 0s, then each inverter will try to force its output to be 1. If all inverters managed to do so, then all inverters' inputs will become 1s, and each inverter will try to force its output to be 0. But all 0s or all 1s states are not stable as they contradict with the functionality of the inverters, which can only be stable in two states "01010101" or "10101010". As the inverters form a feedback loop, they affect each other, and since practically it is impossible to have identical inverters, these slight variations between them will cause some of the inverters to have more drive than others causing oscillations until they eventually reach one of the two stable states or don't stabilize in a reasonable time.

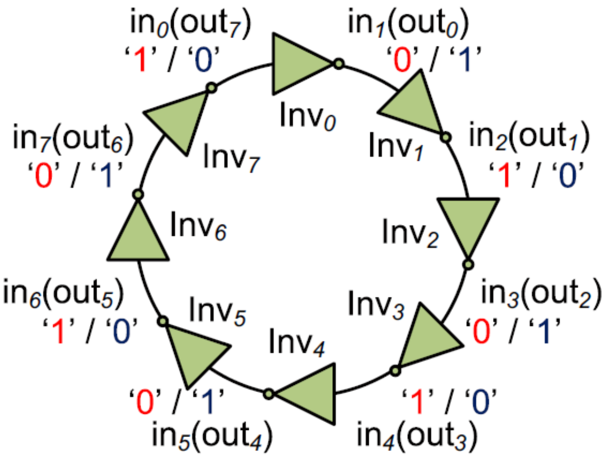


Figure 3.4: Even ring of inverters [13]

The described even ring is considered a weak PUF as it gives only one response, with no challenges introduced. In order to have a strong PUF out of this weak PUF, the architecture in Figure 3.5 is used. For each inverting stage in the ring, there is a block containing two inverters instead of one, associated with a MUX and a DE-MUX controlled by a challenge bit to select which inverter contributes to the ring. To accommodate initialization, the inverters are replaced with either NOR gates or NAND gates with one of the inputs connected to a reset signal. This way all the inverters could be reset and initialized to 0 simultaneously before the PUF action takes effect. With this architecture, with N stages of the mentioned inverting block associated with N-bit challenges, there would exist 2^N different rings each should ideally yield to an independent and random response. The 1-bit response is usually the output of one inverting stage after a fixed time.

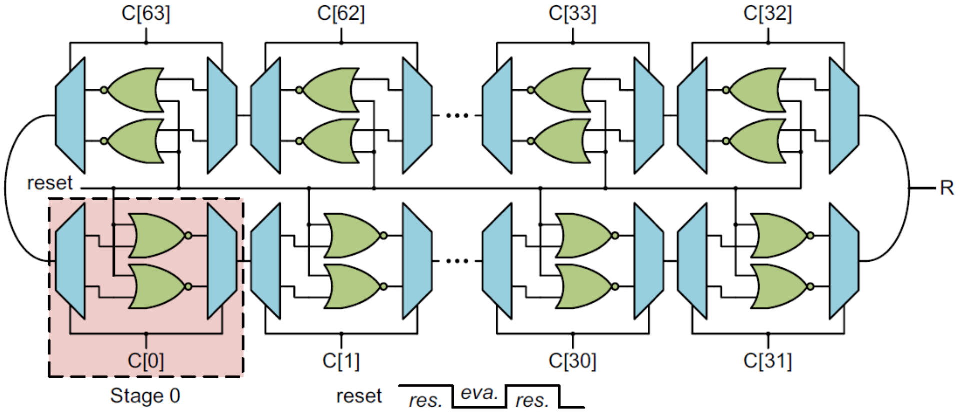


Figure 3.5: BR PUF with 64 stages [13]

Another variant for the BR PUF was introduced in [57] as the TBR PUF. In the BR PUF, depending on the challenge bits, one inverter contributes to the ring and the other is not utilized. However, in the TBR PUF both inverters are utilized while the challenge bits determine the position of the inverter whether it will be in the forward path or the backward path as shown in 3.6.

The main advantage of the BR PUF is being similar to the delay-based PUFs by having a very large challenge-response space, but also its architecture is actually behaving as memory-based PUFs which are resilient to machine learning modeling attacks. So it is expected that the BR PUF would also be resistant to machine learning modeling attacks as well since there is no known mathematical model for the BR PUF as claimed by its developers [13].

However, BR PUF implementations on FPGA were reported to have particular bits of the challenges to have larger influence on the responses than other challenge bits [75] [25] leading to the success of some modelling attacks against both BR and TBR PUFs as in [57], [75] and [25].

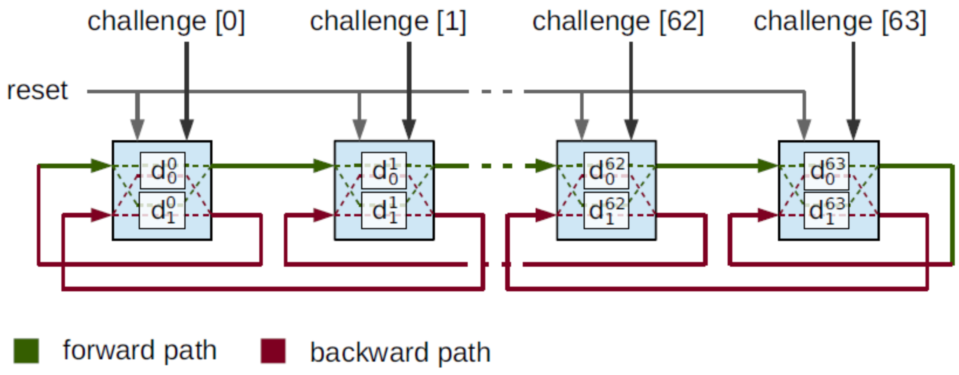


Figure 3.6: TBR PUF with 64 stages [57]

3.4 XOR PUFs

To make PUFs more resilient against machine learning modeling attacks, the PUF's response could be obfuscated by XORing multiple PUF instances' outputs as depicted in Figure 3.7 to produce a final response. Which adds non-linearity to the system, making modeling attacks more difficult [69].

While XOR PUFs showed more resistance against modeling attacks, however, in [56], 4 and 5-input XOR arbiter PUFs with 64 and 128-bit challenges were attacked using Linear Regression (LR). Also in [74] 3-input XOR 32-bit and 64-bit BR PUF were modelled using Support Vector Machine (SVM). However, the attack failed to model 4-input XOR BR PUFs and even 3-input XOR BR PUFs with challenge bits more than 64.

3.5 Strong PUFs Characteristics

A strong PUF should achieve three main properties: reproducability, unpredictability and uniqueness [53] [12].

- **Reproducability** of a PUF is to have, with a high probability, similar responses resulting from the same challenges when measured at different times under possibly varying operating conditions. This is an intra-chip property which means it is defined per each PUF instance considering its own CRPs. A strong PUF should be reproducible or otherwise the responses would be noisy, making it unreliable for authentication purposes.

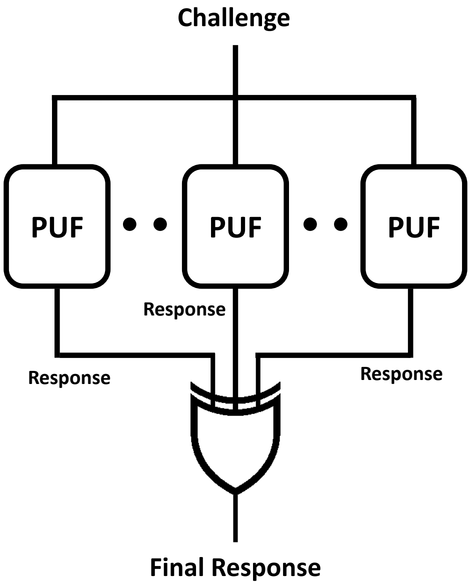


Figure 3.7: XOR PUF

- **Unpredictability** is the fact that the responses are expected to be random in relation to the challenges. Even with the knowledge of some CRPs of a PUF instance, unobserved responses should remain sufficiently random with no way to predict them through their corresponding challenges. Unpredictability is also an intra-chip property applying to each PUF instance individually. It is very important for a PUF to be unpredictable, or an adversary will be able to model the PUF with the knowledge of some CRPs, enabling the adversary to clone the PUF and compromise authentication.

- **Uniqueness** is the capability of identifying each PUF instance from a set of PUF instances that have gone through the same manufacturing process. This requires that responses resulting from the same challenges on different PUF instances should be dissimilar with high probability. Unlike reproducibility and unpredictability, uniqueness is an inter-chip property defined over the class of all PUF instances. Uniqueness is the property that actually makes such architectures be called PUFs by providing an identity and fingerprint to each chip.

3.6 BR/TBR PUFs Previous Research

The BR PUF was introduced by Chen et al. in [14] as a promising candidate for strong PUFs. The bistable ring consisting of an even number of inverters was introduced as a weak PUF, however, with introducing the architecture depicted in Figure 3.5 a strong PUF with exponential CRPs is established. FPGA implementations were used to show the quality of the BR PUF. Experiments were run on eight Xilinx Virtex-II Pro FPGA boards, implementing 32, 64, 128-bit BR PUFs. However, no details were mentioned on the layout used for implementing the PUF.

To measure stabilization, the ring state was examined 28 times at different time points, resulting in an average settling time over the eight boards of 5.26 μs, 10.78 μs and 23.09 μs to reach 90% of stabilized responses for the 32, 64 and 128-bit BR PUFs respectively. However, the maximum settling time among the eight chips were reported to be 8.44 μs, 22.25 μs and 37.20 μs for the 32, 64 and 128-bit BR PUFs respectively.

The implementation of the 64-bit BR PUF showed a very low inter-chip hamming distance of 14.8% which is far away from the ideal value of 50%. However, a good intra-chip hamming distance of 0.8% was reported. The interesting finding in this paper is that the responses with short settling times were found to be more reliable, however, constant among different chips. while responses with long settling times showed more uniqueness of the chip but less reliable. Therefore suggesting to use only the CRPs with longer settling times (between 35 μs and 47 μs) as a simple solution to achieve better identification.

Since the paper in [14] relied on FPGAs to show the characteristics of the BR PUF, which seemed to be layout biased. A sequel in [15] by Chen et al. used transistor-level SPICE simulations to further investigate the BR PUF characteristics. The simulations used the nominal process variation, mismatch and noise models from industry to predict the characteristics of an Application Specific Integrated Circuit (ASIC) implementation of the BR PUF. Due to the relatively large number of transistors of one BR PUF stage (16 transistors) and the fact that the BR PUF is a loop structure, simulations were very slow, thus only a 32-bit implementation was used.

Based on experiments on 15 instances, the results show that 90% of the responses stabilize in 2.315 μs. Inter-chip hamming distances ranging from 35.7% to 67.6% with a 50.9% average were reported, which is acceptable as the ideal value is 50%. Also, intra-chip hamming distance ranging from 0.2% to 3.0%, with an average of 1.3% were reported with the use of ten different sets of random transient noise in the simulations.

Unlike the results presented in [14], the settling time of responses didn't affect the uniqueness of the PUF with the ASIC simulation, but uniqueness is presented through all settling times. These results show that the BR PUF actually has the potential of a strong PUF despite the biased characteristics experienced in [14] which might have been an FPGA implementation issue.

In [76], Yamamoto et al. tried to propose methods for evaluating the predictability of PUF responses inspired by differential and linear cryptanalysis, and applied these methods on the BR PUF. The paper tries to investigate the correlation between the responses generated by challenges that have small hamming distance as well as the correlation between the responses generated by challenges whose particular bits are forced to be 1 or 0.

The experiments run on four Xilinx Spartan-6 FPGAs with custom layout showed that 88% of the challenges with only 1 bit of hamming distance lead to the same response which is a very huge percentage. This percentage decreases with the increase of the challenges hamming distance to reach 66.5% when 16 bits are different which is still not negligible. This correlation allows an attacker to predict unknown responses from some known CRPs given small hamming distances between the challenges. However, this correlation tends to be less for the CRPs with longer settling time, being 68.1% when only 1 challenge bit is different and only 55% when 16 bits are different.

Another type of correlation reported was that some NOR gates have a larger influence on the responses than others. It was observed that five particular bits in the challenge being fixed with some values lead to a probability of 71.4% of having a response of 1. Other combinations of four bits in the challenge can lead to a probability of around 69% of the responses being 1. The influence decreases with the number of bits combined reaching 54.5% for particular single bits. These correlations also make unknown responses more predictable using a set of known CRPs if the influential bits are figured out.

In [58], Schuster et al. introduced the TBR PUF as an alternative implementation of the BR PUF, along with analysis using Artificial Neural Network (ANN) on FPGA implementations of both BR and TBR PUFs. The 64-bit PUFs implemented on 20 different Digilent NEXYS boards with Xilinx Spartan-6 FPGA showed noise ranging from 2% to 19% and bias deviation ranging from 18% to 48% for the BR PUF. However, for the TBR PUF the noise was reported to range from 7% to 23% with a bias deviation ranging from 1% to 38%. An interesting note is that while for the TBR PUF the instances are distributed among both characteristics, meaning that some good TBR PUF instances exist, for example one having moderate noise of 9% and low bias of 1% as well. However, for the

BR PUF the instances with low noise have very high bias deviation, and those with lower bias deviation have very high noise. These results showed biased BR PUF implementations, however for TBR PUFs some were biased and some were not. But this is not enough to say that TBR PUFs are better than BR PUFs since these results are specific to these implementations and show bias anyway.

The neural network used to analyze both PUFs consisted of 64 input neurons representing the 64 bits of the challenge in addition to another bias neuron constantly set to 1. The input layer was fully connected to the output layer which consists of two neurons representing a one-hot encoding since it is a classification problem with 0 or 1 outputs. Five-fold cross-validation was used for the learning process. The results of this modelling attack yielded to a successful linear approximation for both BR and TBR PUFs with learning errors ranging from 0% to 20% for both PUFs. The main difference is that for the BR PUF the majority of instances (14 instances) were approximated with learning errors below 10%. However, for the TBR PUF the majority of instances (16 instances) were approximated with learning errors above 10%. Which might indicate that the presented TBR PUF implementations show better performance against machine learning linear approximation than the presented BR PUF implementations. However, that does not give any superiority of the TBR PUF over the BR PUF, since their presented implementations showed favorable characteristics of the TBR PUF from the first place, which as commented before is more of a layout bias specific to their implementation.

Interestingly, it was found after the training of the neural networks that there were no patterns on the weights of each of the input neurons except for the weight of the bias neuron, meaning that the ANN acted naively relying mainly on the bias regardless of the challenge bits values. Which makes sense for this case since the PUFs implementations were biased to begin with.

Ganji et al. in [25] and its extension [24] aimed to provide a provable framework for ML attacks against PUFs even those without a known mathematical model. As an example, the papers apply their technique on the BR and TBR PUF family being claimed not to have a know mathematical model [14]. The paper in [25] starts by showing theoretically that for all PUFs when presented as Boolean functions, it holds that their degree as $\mathbb{F}_2$-polynomial is greater than one. Leading to the conclusion that not all challenge bits have an equal influence on the PUF responses, thus could be Probably Approximately Correct (PAC) learned. While the phenomenon of having particular influential bits has been observed in many previous implementations, but this paper provided a mathematical proof of the existence of such influential bits that holds for every PUF.

Further experiments were conducted on BR and TBR PUFs implementations on Altera Cyclone IV FPGAs. Using 30000 CRPs, it was found that in all instances, at least one influential bit existed, while the maximum number of influential bits (the combination of bits in a pattern that gives 100% probability of response being 1 or 0) is fixed, being only 5 for 64-bit BR and TBR PUFs. The characteristics of the implemented PUFs were not detailed, only it was mentioned that they were originally highly biased thus different manual placement and routing configurations were applied without describing any. However, it is worth mentioning that in this work the authors took the PUFs' readings at different points of time, and if responses were not the same for all times, they were considered unstable. While one can argue that BR PUFs are known to have a period of time till stabilization. And according to the previous work [14] [76], they showed that the CRPs with short settling times are actually biased due to the implementation layout, while the ones with long CRPs are the ones exhibiting the uniqueness of the PUF. Thus, excluding the CRPs with long settling times raises doubts about the accuracy of the resulted data.

Applying three different weak learner models (monomials, decision trees, and decision lists) boosted with the adaptive boosting algorithm, with training sets of 100 and 1000 CRPs of total 30000 CRPs, the authors were able to demonstrate successful modeling for both BR and TBR PUFs with variant accuracy. Results showed that without boosting, decision lists were able to predict the BR PUF behavior with 67.24% and 84.59% with the use of 100 and 1000 training set sizes respectively. Decision lists also were able to predict the implemented BR PUFs with higher accuracy of both 74.84% and 84.34% with the use of 100 and 1000 training set sizes respectively. With the help of the boosting algorithm, the accuracy increased dramatically reaching 98.32% for the BR PUF with 50 boosting iterations using 1000 CRPs as a training set. As well as reaching 99.37% for the BR PUF with the same parameters.

These results show that the implemented BR PUF is more resistant than the TBR PUF against the modelling attacks which contradicts with the data presented in [58], which as previously mentioned is mainly an implementation bias. And there is no evidence in the literature that either BR or the TBR PUF is better than the other since all the implementations in the literature are FPGA based, which cannot be fully symmetric and with no bias.

The work in [74] by Xu et al. evaluated the BR and TBR PUFs using SVM. Moreover, it introduced the XORing of multiple PUF instances' outputs to enhance the security as well as the uniformity of the BR PUF. The intuition behind their modeling attack is that the BR PUF's response could be predicted as a summation of weights corresponding to each stage. The weight of each stage could be represented as the difference between the pull-up strength and the pull-down strength of each NOR gate. If the sum of all stages' weights is positive then that would yield to a response of 1, otherwise, it would be a response of 0. Using that model with SVM of a linear kernel and applying that on BR PUF implementations of different sizes (32, 64, 128 and 256 bits) on 8 Xilinx Spartan-6 FPGA boards, successful modelling exceeding 95% prediction accuracy was performed on all PUF sizes. To reach 95% of prediction accuracy, 780 CRPs were needed as a training set in the case of 32-bit BR PUF. Similarly, 1350, 2400 and 6000 CRPs were needed to achieve 95% prediction accuracy on 64, 128 and 256-bit BR PUFs respectively.

The authors showed that the TBR PUF could be predicted using the same model, and applied the same attack reaching 95% prediction accuracy using only 420, 750, 1250 and 2700 CRPs as training sets for the 32, 64, 128 and 256-bit TBR PUFs respectively. Which seems even easier than modelling the BR PUF.

As results on both BR and TBR PUFs, showed that individual instances are not secure, the authors proposed XORing multiple PUF instances' outputs to obfuscate the individual PUF responses and add some non-linearity, inspired by the same approach on the arbiter PUF. The same modeling attack was performed on XORed BR PUFs using SVM but this time with polynomial kernel corresponding to the number of instances XORed. The modeling attempt was only successful on 2-input XOR of 32, 64 and 128-bit PUF sizes, failing on the 256-bit size. And succeeded as well on 3-input XOR only with 32 and 64-bit PUF sizes, failing on both 128 and 256-bit sizes. And never succeeded with 4-input XOR of any size. Which shows a real improvement on the BR PUF security.

Not only improvement on the PUF's security but also an improvement on the PUF's bias was observed. As reported, the single PUFs were highly biased having the best two instances with 40% and 60% bias, while the rest of the instances are above 70%. However, after XORing, the bias gets nearer to the ideal value of 50%. For example, all 4-input XOR PUFs had biases between 40% and 60%. Reaching around 50% with 8-input XOR. Despite these improvements, the drawback of the XOR PUF was the increased noise since if one PUF instance is not stable then the XORed value will not be stable too. Thus, an increased noise over a single BR PUF.

In summary, the BR PUF was introduced in [14] as a promising architecture, followed by the introduction of its variant the TBR PUF in [58] to improve its characteristics. Several modelling attacks in [76], [58], and [25] revealed the presence of specific challenge bits that have more influence on the PUF response than others. The XOR BR PUF was studied in [74] and showed that 4-input XOR BR PUFs are resilience against modelling attacks.

Chapter 4

EMFI Experimental Setup

This chapter goes through the setup used for the EMFI experiments, its hardware components, software components, and the EM pulse characteristics in Section 4.1. In addition, more information about the target processor is provided in Section 4.2.

4.1 EMFI Setup

As shown in Figure 4.1, the EMFI station utilizes a burst power station, an EM probe, a probe tip, a CNC machine, an oscilloscope, a power supply, the target board and a Personal Computer (PC).

The burst power station controlled by the PC generates a low-rise-time voltage pulse to the EM probe which generates an EM pulse delivered by the probe tip placed over the target chip. To control the timing of the generated pulse, prior to the target instruction, the target processor sends a trigger signal to the burst power station which responses with the pulse after a specific delay. The oscilloscope is used to measure the delay from the trigger to the pulse, the delay's jitter as well as the pulse rise time. The CNC machine worked as a movable XYZ stage to place the probe tip over the target chip. To control the supplied voltage, the voltage regulator in the target board was removed and the core's VDD pin was connected to the power supply. For frequency control, the on-chip Phase-Locked Loop (PLL) was used to provide frequencies up to 384 MHz. The PC also interfaces with the target board via Universal Serial Bus (USB) for code upload and debugging.

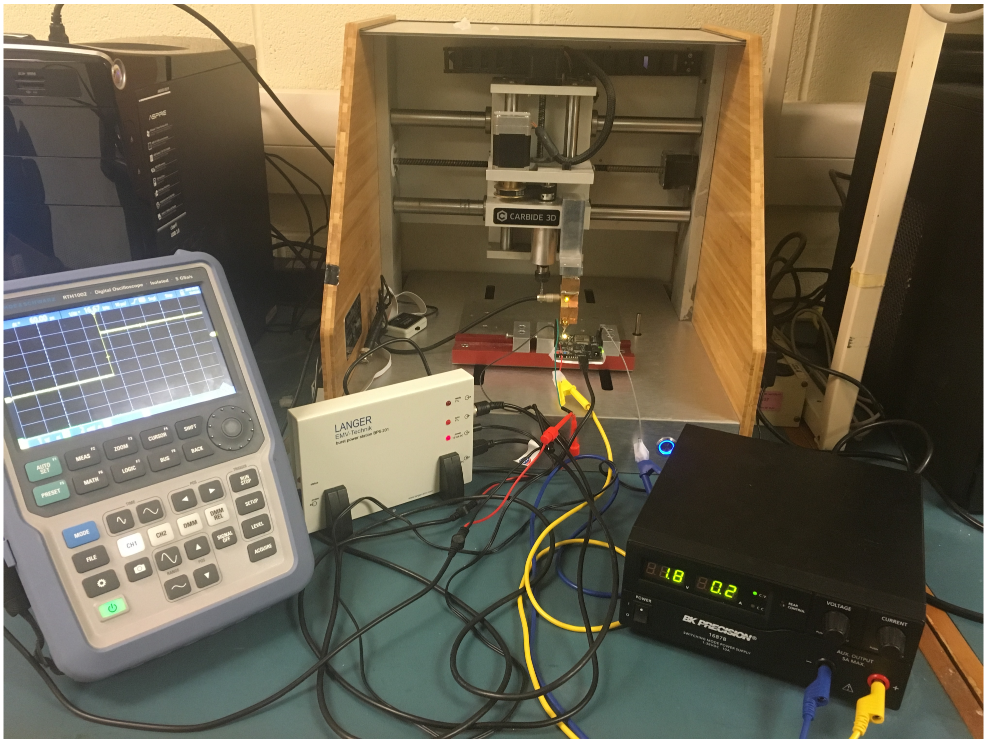


Figure 4.1: EMFI hardware setup

4.1.1 Hardware

A detailed description of each hardware component is as follows:

- **EM pulse generation system:**

  The burst power station BPS-201 along with the ICI probe from Langer EMV-Technik [38] form the EM pulse generation system that is able to generate EM pulses with amplitude ranging from 130 V to 180 V and a minimum delay between two consecutive pulses of 100 μs. The system can generate pulses with a fixed frequency or upon a trigger signal, which is the mode used.

- **EM probe tip:**

  A 1 mm in diameter home-made probe tip is attached to the EM probe and placed approximately 200 μm over the chip as displayed in Figure 4.2

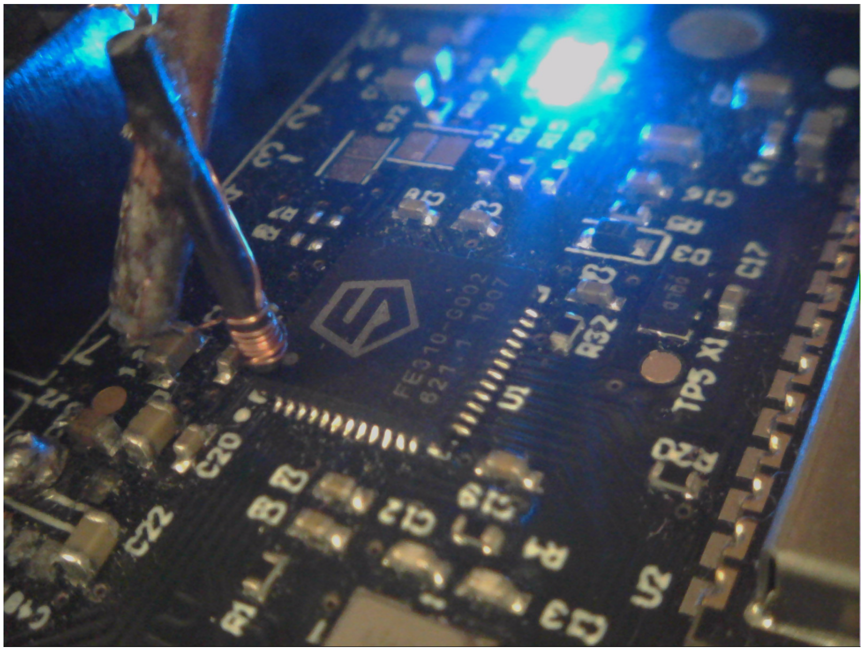


Figure 4.2: Probe tip over the chip

- **Power supply**.

  The 1687B power supply from BK Precision [10] comes with the feature of remote access using USB, which was beneficial for experiment automation by controlling the voltage supplied to the core using scripts on the PC side.

- **Oscilloscope**

  The RTH-1002 handheld digital oscilloscope from Rohde & Schwarz [54] is a two Channel, 60-MHz oscilloscope, with an acquisition rate of up to 50000 waveforms per second and 5G sample per second sampling rate.

- **CNC machine**

  The Nomad 883 Pro from Carbide 3D [11] is a low-cost CNC machine with a resolution of .0005 in (0.0127 mm).

4.1.2 Software

To control the hardware components and to facilitate experiment automation, three components on the PC side were involved: BPS-201 client Graphical User Interface (GUI) to control the EM pulse generation system, Freedom E SDK as a tool-chain for the target and scripts to automate the whole process.

- **BPS-201 client GUI:**

  Langer EMV-Technik provides BPS-201 client GUI to control its BPS-201 system. The software allows for many parameters to be configured, such as pulse voltage, pulse polarity, external trigger activation, and trigger-to-pulse delay.

  For all experiments conducted in this research, the parameters shown in Figure 4.3 were fixed, except for the pulse voltage. The single pulse mode was used with positive polarity for the pulse. The external trigger was enabled on the rising edge, while choosing the minimum trigger delay of 130 ns, which is the delay from the trigger to the pulse sent to the probe, not accounting the delay caused by the probe itself.

- **Freedom E SDK:**

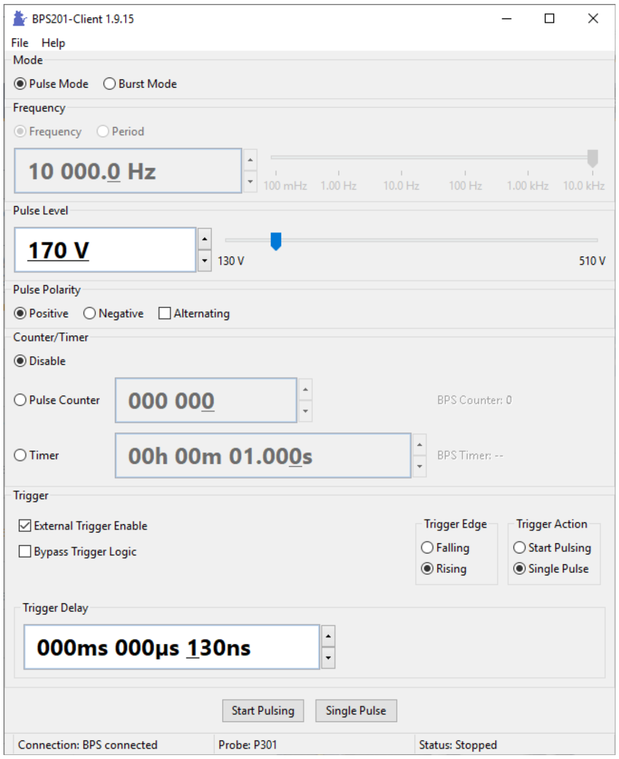
  Freedom E SDK is a Linux-based Software Development Kit (SDK) developed by SiFive [61] to facilitate building and uploading RISC-V codes to its supported platforms. It uses RISC-V GNU tool-chain beside Board Support Packages (BSPs) designed for its platforms, alongside with scripts and makefiles for builds and uploads.

- **Automation scripts:**

  Python scripts were designed to control the CNC machine and power supply via USB as for each experiment's requirements. Also, Bash scripts were used to modify the codes if needed, build and upload them to the target. Bash scripts were also used to reset the target every run and retrieve the content of the register file of the target after every run using JTAG.

4.1.3 EM Pulse Characteristics

It is very important to measure the characteristics of the pulse generated by the probe, to better understand its the effect on the faults injected and the corresponding timing analysis.



Mode

● Pulse Mode ○ Burst Mode

Frequency

● Frequency ○ Period

10 000.0 Hz

Pulse Level

170 V

Pulse Polarity

● Positive ○ Negative □ Alternating

Counter/Timer

● Disable

○ Pulse Counter 000 000 BPS Counter: 0

○ Timer 00h 00m 01.000s BPS Timer: --

Trigger

☑ External Trigger Enable

□ Bypass Trigger Logic

Trigger Edge

○ Falling

● Rising

Trigger Action

○ Start Pulsing

● Single Pulse

Trigger Delay

000ms 000µs 130ns

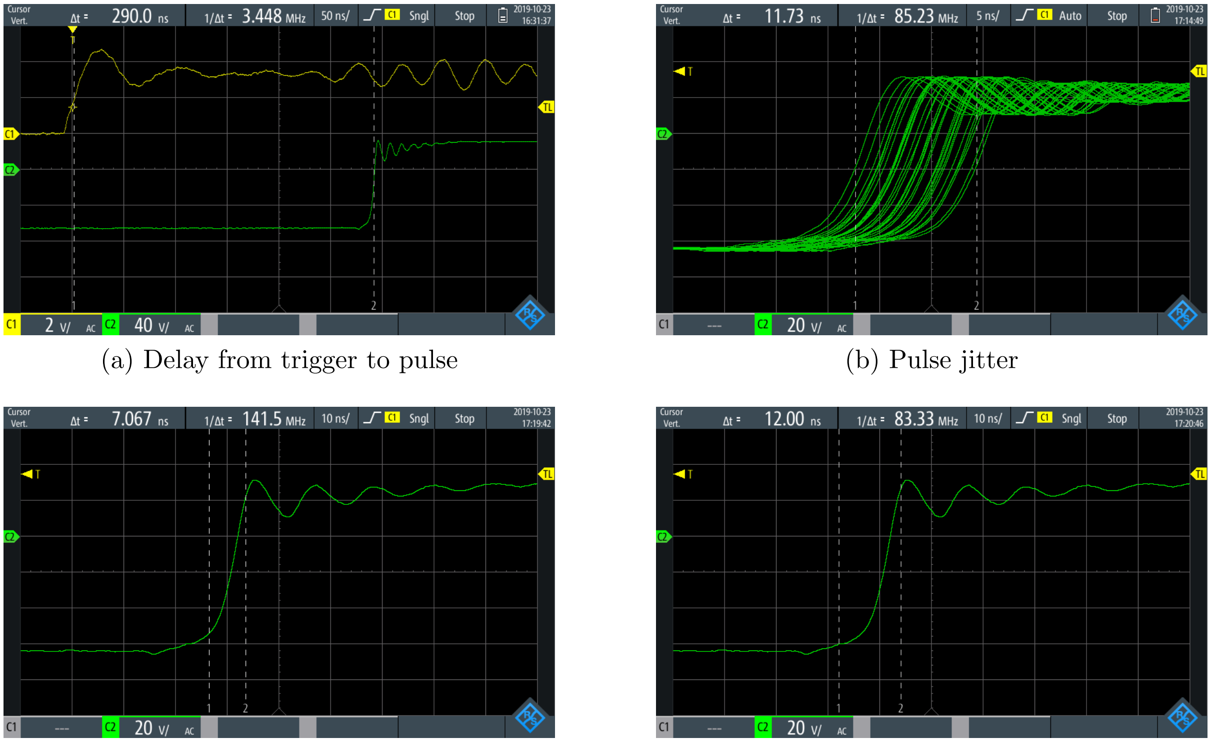
Connection: BPS connected Probe: P301 Status: Stopped

Figure 4.3: BPS-201 client GUI

The oscilloscope was used with channel one connected to the trigger signal from the target processor, and channel two connected to the EM probe. All measurements were taken with a pulse voltage of 170 V and a trigger delay of 130 ns.

As Figure 4.4a shows, the average delay from the trigger to the pulse out of the probe is 290 ns. Unfortunately, the delay actually ranges from 284 ns to 296 ns due to the 12 ns jitter created by the system as displayed in Figure 4.4b. The delay measurements were taken from the time the trigger signal reached 50% of its amplitude to the time the pulse signal reached 50% of its amplitude.

Figure 4.4c shows the rise time of the pulse signal from 10% to 90% of its amplitude to be 7 ns. However, the rise time would be 12 ns if measured from 5% to 95% of the signal's amplitude as in 4.4d.



(a) Delay from trigger to pulse

(b) Pulse jitter

(c) Rise time 10%-90%

(d) Rise time 5%-95%

Figure 4.4: EM pulse characteristics

4.2 The Target Device

For the experiments conducted in this research, HiFive1 Rev B development board from SiFive [61] was used. The board shown in Figure 4.5 features SiFive's FE310-G002 chip as the main SoC, beside a 32-Mbit (8-MB) Serial Peripheral Interface (SPI) flash memory, as well as a Segger J-Link OB module [59] which bridges USB to JTAG [64].

The FE310-G002 chip is built around an E31 core complex, a General Purpose Input Output (GPIO) complex, in addition to a dedicated Quad Serial Peripheral Interface (QSPI) controller to interface with the flash memory with an execute-in-place mode, as well as a clock generation module containing a programmable PLL. The 48-pin chip is fabricated in TSMC CL018G 180 nm process and packaged in a Quad Flat No-leads (QFN) package. It uses 1.8 V to supply the core and 3.3 V for the Input Output (IO) pads [63].

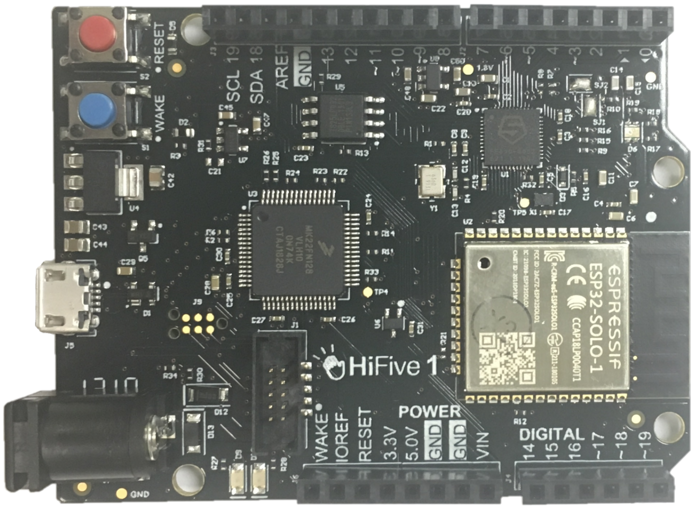


Figure 4.5: SiFive board

The E31 core on FE310-G002 is a high-performance single-issue in-order execution five-stage pipeline processor. The five stages are: instruction fetch, instruction decode and register fetch, execute, data memory access, and register writeback. The core supports standard Multiply, Atomic, and Compressed RISC-V extensions (RV32IMAC ISA) [62].

The core features a 16-KB Data Tightly Integrated Memory (DTIM) as well as a 16-KB two-way set-associative L1 cache. The instruction cache can be configured to have one way as an Instruction Tightly Integrated Memory (ITIM). Since fetching an instruction from ITIM takes only one clock cycle, the same as an instruction cache hit, ITIM is considered a more predictable and a higher performance instruction fetch option compared to fetching from the flash memory via SPI. However, for the DTIM the access latency from the core is two clock cycles for full words [63].

For improved performance, the core contains a branch prediction hardware which comprises a 28-entry Branch Target Buffer (BTB) which predicts the address of taken branches, a 512-entry Branch History Table (BHT) which predicts if a conditional branch is to be taken or not, and a six-entry Return Address Stack (RAS) which predicts the address of functions/routines returns. With this additional hardware, a correctly predicted control-flow instruction has no misprediction penalty, however, a mispredicted one will incur a three-cycle penalty [63].

Chapter 5

EMFI Experiments and Results

Five different types of experiments were performed on the RISC-V core. In this chapter, a detailed illustration of the experiments is provided, as well as their results. The objectives of these experiments are as follows:

1. To identify the best location over the chip where faults are more likely to be injected, which is detailed in Section 5.1.
2. To examine the various conditions such as supplied voltage and clock frequency on the faults susceptibility, as illustrated in Section 5.2.
3. To perform timing analysis on when to trigger the EM system for faults to be injected, which is discussed in Section 5.3.
4. To explore if many instructions can be faulted at once, as depicted in Section 5.4.
5. To analyze the effect of the fault injection on individual instructions, which is provided in Section 5.5.

5.1 Identifying the Best Spot for EMFI

To identify the best spot over the chip to inject faults, a scan over the chip using the CNC machine holding the probe tip was performed, and the numbers of faults injected per spot were recorded as developed in [40].

The code used for this experiment as illustrated in Code 5.1 is based on a main loop that iterates for 10 times. For each iteration, the EM system trigger is asserted and multiple ADDI instructions are performed. The window of the ADDI instructions (300 instructions) is large enough so that regardless of the accurate timing of the applied EM pulse, one of the instructions should be exposed to the EM pulse. If one of the instructions is faulted, then the final sum will be different from the expected correct sum, hence an injected fault is detected. Another scenario could be that the faulted instruction is corrupted causing an exception, which is detectable as well by the exception routine. By the end of the program, register x19 should contain one of the following three values:

- 0xAA: which means no fault was injected
- 0xBB: which means sum is not correct, thus a fault was successfully injected.
- 0xCC: which means there was an exception raised causing the processor to enter the trap vector. Exceptions could be due to invalid instructions or invalid addresses.

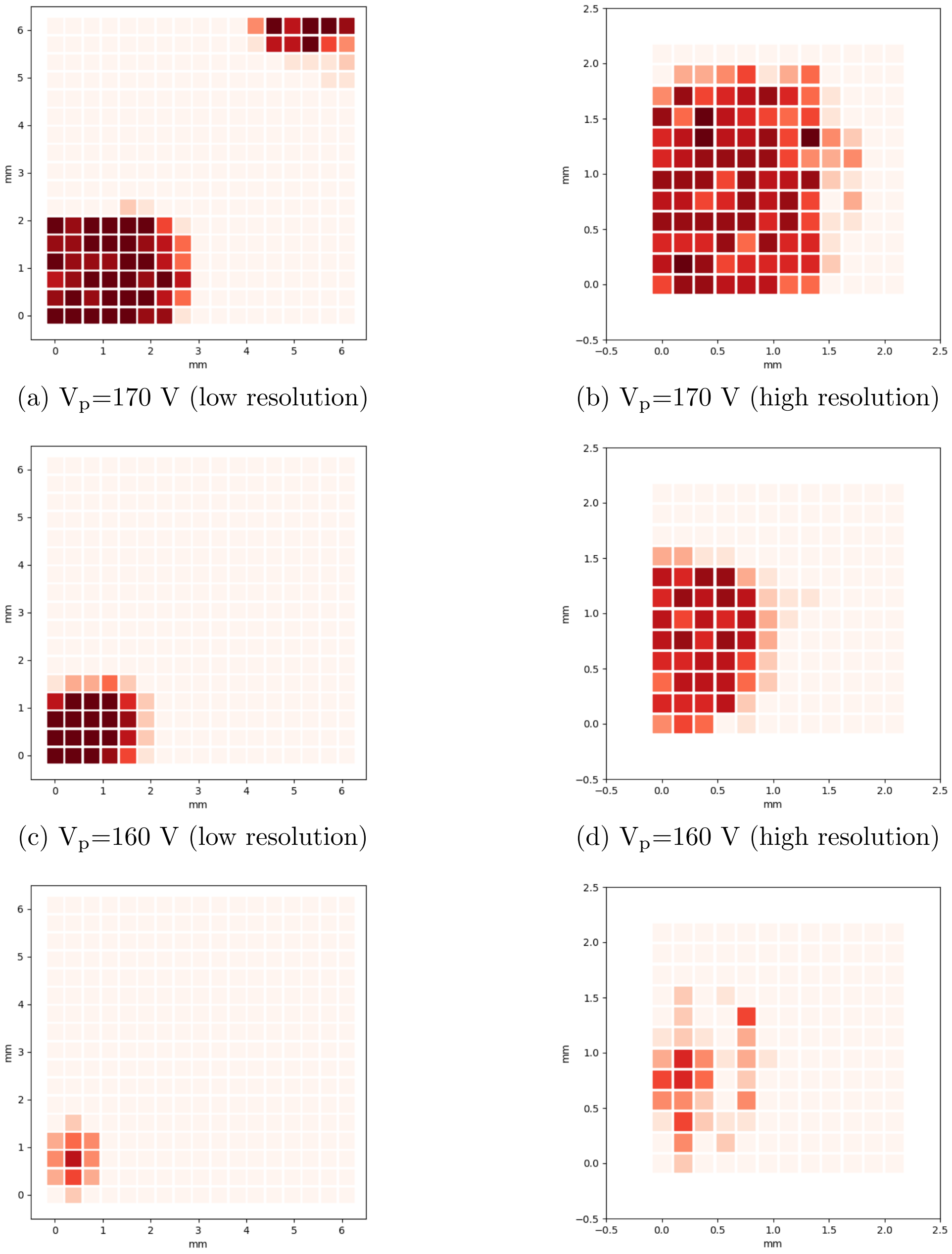
After each run, the JTAG is used to halt the processor and access all the general-purpose registers to be logged in files for further analysis.

Code 5.1: Test program to capture any fault injected to any of multiple ADDI instructions

```
main:

    #Initializations
    li x22,10                   #Number of rounds for the main loop 10
    addi x19,x0,0x99            #Initializes Error Code to be 0x99

    #Main Loop
    LOOP:

        li x16,301              #Correct Sum
        li x17,1                #Initializes x17 to be 1

        (Assert the trigger)    #Sets an IO to assert the trigger

        addi x17,x17,1
              .
              .
        (300 times in total)
              .
              .
        addi x17,x17,1

        sub x14,x16,x17           #x14=x16 - x17
        bne x14,x0,FAULT          #If x14 not 0, jump to FAULT

        (De-Assert the trigger) #Clears an IO to de-assert the trigger

        addi x22,x22,-1           #Decrements rounds counter by 1
        bne x22,x0,LOOP           #keeps in loop until counter is 0

    #No Fault Injected
    NO_FAULT:
        addi x19,x0,0xAA          #Error Code 0xAA for no fault
        j NO_FAULT

    #Fault Injected
    FAULT:
        addi x19,x0,0xBB          #Error Code 0xBB for fault injected
        j FAULT

    #Exception Fault
    early_trap_vector:
        csrr x5,mcause            #Exception code for the exception cause
        csrr x6,mepc              #PC when the exception took place
        csrr x7,mtval             #The corrupted instruction/address
        addi x19,x0,0xCC          #Error Code 0xCC for exception fault
        j early_trap_vector
```

Utilizing the aforementioned code, six different scans over the chip with variant parameters were performed. For each experiment, the code was run 10 times on each spot (this is not to be mistaken for the 10 iterations in the code, as the whole experiment was applied for 10 rounds). The numbers of faults injected on each coordinate were used to draw shmoo plots to visualize the best location for injecting the faults as displayed in Figure 5.1. For all experiments, the maximum functional frequency defined by the manufacturer (320 MHz) and the nominal supply voltage (1.8 V) were used.

In the beginning, with an EM pulse amplitude of 170 V, a scan over the 6 × 6 mm chip with steps of 381 μm was performed. This yielded into 17 × 17 spots covered. Figure 5.1a shows one large spot to the bottom left corner of the chip where faults can be injected. Also, it shows a smaller spot on the top right corner where faults can be injected. A similar experiment targeting the bottom left ninth of the chip (2 × 2 mm) but with a



(a) V_p=170 V (low resolution)

(b) V_p=170 V (high resolution)

(c) V_p=160 V (low resolution)

(d) V_p=160 V (high resolution)

(e) V_p=150 V (low resolution)

(f) V_p=150 V (high resolution)

Figure 5.1: Shmoo plots for the number of faults injected over the chip using different EM pulse amplitudes and resolutions. Keeping frequency = 320 MHz and VDD = 1.8 V

higher resolution (a step of 190 μm) was performed, but as shown in Figure 5.1b still that did not show the exact best spot to inject faults.

To narrow down the spots where faults can be injected, the EM pulse amplitude was reduced to 160 V, and the same experiment with the higher resolution was performed. While in Figures 5.1c and 5.1d the spots seem to narrow down, but still not accurate.

Eventually, with the use of an EM pulse amplitude of 150 V, it was clear which spot exactly over the chip is the best to inject faults (which is 190 μm to the right of the left edge of the chip and 950 μm above the bottom edge of the chip) as displayed in Figures 5.1e and 5.1f.

5.2 Examining the Various Conditions for EMFI

In this section, experiments examining the effect of both clock frequency as well as voltage supplied to the core on EMFI are presented similar to the approach in [40].

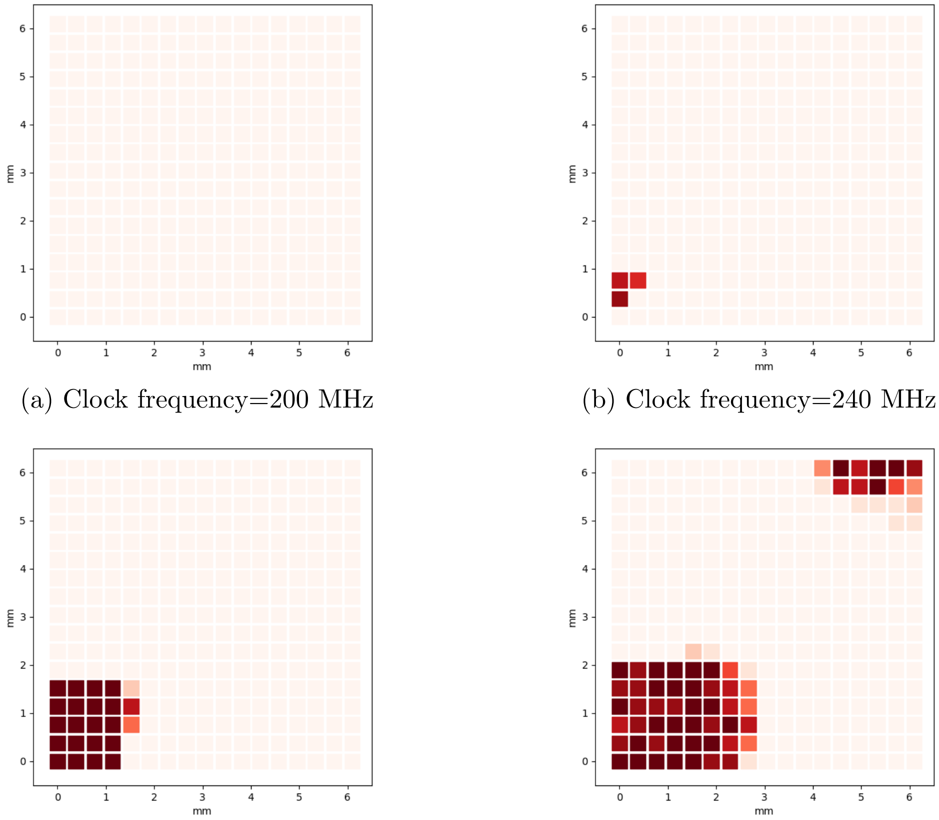
5.2.1 Effect of the Clock Frequency

To study the effect of clock frequency on the fault injection, the same program in Code 5.1 was utilized with different clock frequencies (200 MHz, 240 MHz, 280 MHz, and 320 MHz) with a scan over the chip with steps of 381 μm. For all experiments, the core was operating on the nominal supply voltage (1.8 V) and the EM pulse amplitude used was 170 V.

As Figure 5.2 shows, the higher the frequency the more faults are injected. As operating with 200 MHz or less, no faults could be injected. And with the maximum functional frequency (320 MHz), even different spots start to be fault susceptible.

5.2.2 Effect of the Supplied Voltage

The same program in Code 5.1 was used again but this time with varying the voltage supplied to the core. Three supply voltages were used (1.8 V, 1.7 V, and 1.6 V) on two different frequencies (240 MHz and 280 MHz) with a scan over the chip with steps of 381 μm. For all experiments, the EM pulse amplitude used was 170 V. With no EMFI, the core was empirically verified to be functional operating on the three different supply voltages at the listed clock frequencies.



(a) Clock frequency=200 MHz

(b) Clock frequency=240 MHz

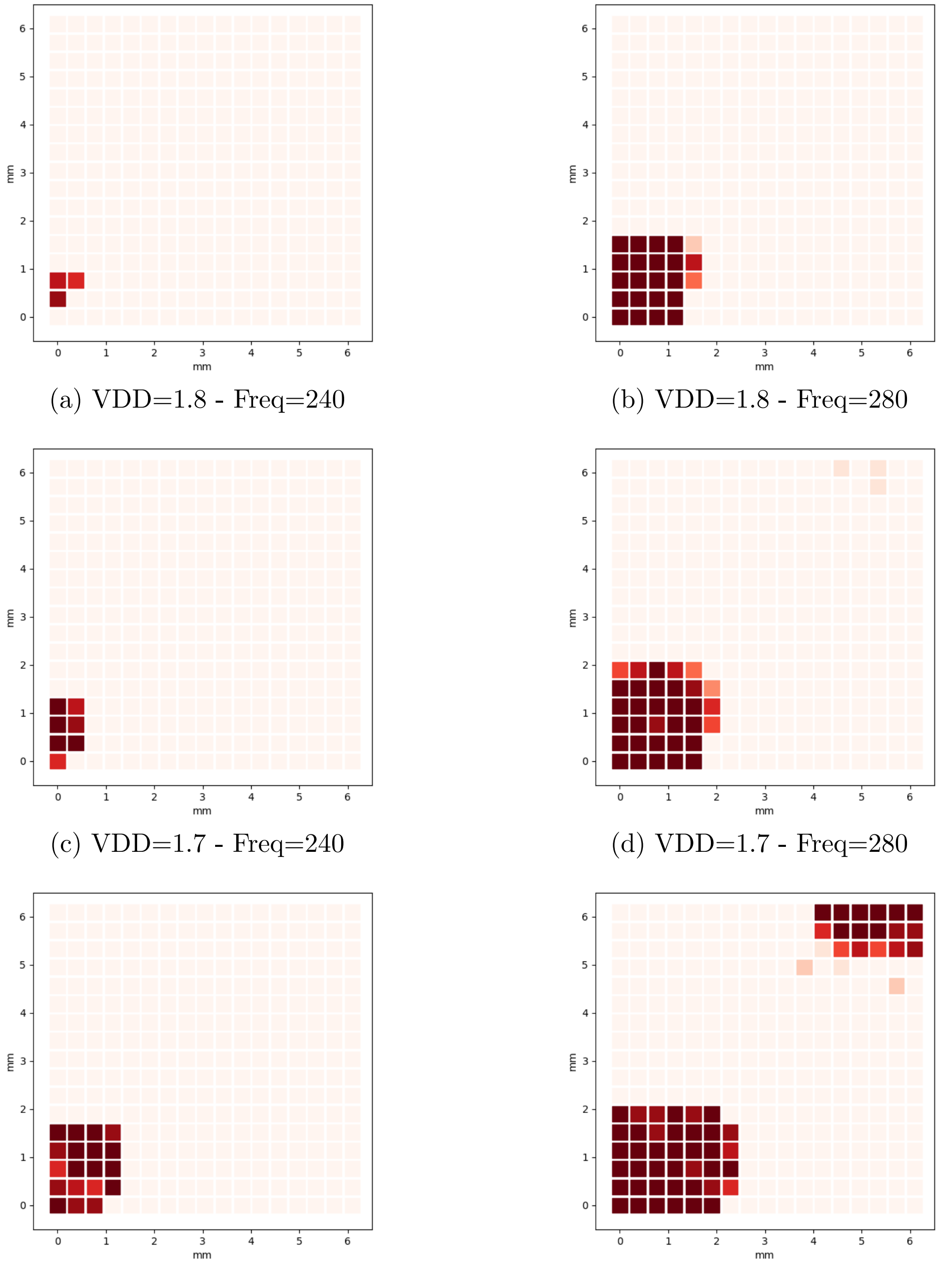
(c) Clock frequency=280 MHz

(d) Clock frequency=320 MHz

Figure 5.2: Shmoo plots for the number of faults injected over the chip using different frequencies. Keeping pulse amplitude = 170 V and VDD = 1.8 V

The variation in the numbers of faults injected in Figures 5.3a, 5.3c and 5.3e (all operating on 240 MHz) provides evidence that by decreasing the supplied voltage, the core is more susceptible to fault injection. Same goes for Figures 5.3b, 5.3d and 5.3f (all operating on 280 MHz). Moreover, while operating on 1.6 V, the spots on the top right corner appeared with 280 MHz clock frequency, which was only possible with 320 MHz clock frequency if operating on nominal voltage of 1.8 V.

The data visualized in both Figures 5.2 and 5.3 indicates that the processor is more susceptible to EMFI with the increase of clock frequency and with the decrease in supplied voltage, which complies with the charge-based fault model introduced in [40].



(a) VDD=1.8 - Freq=240

(b) VDD=1.8 - Freq=280

(c) VDD=1.7 - Freq=240

(d) VDD=1.7 - Freq=280

(e) VDD=1.6 - Freq=240

(f) VDD=1.6 - Freq=280

Figure 5.3: Shmoo plots for the number of faults injected over the chip using different supplied voltages and frequencies. Keeping pulse amplitude = 170 V

5.3 Timing Analysis of the Target Instruction

This section provides a timing analysis of the code execution on the different types of memories, and explores the best position for the target instruction to inject faults.

5.3.1 Measuring Execution Time on Different Memories

In the previous experiments, as shown in Code 5.1, the main loop is iterated for 10 rounds trying to inject a fault. It was noticed in Table 5.1 that faults never been injected from the first round and they would only start being injected from the second round and up.

Table 5.1: Number of faults injected for each loop iteration in Code 5.1. Pulse amplitude = 170 V, VDD = 1.8 V and frequency = 320 MHz

Round number	Number of faults	Percentage
1	0	0%
2	163	30.2%
3	194	35.9%
4	51	9.4%
5	28	5.2%
6	31	5.7%
7	26	4.8%
8	20	3.7%
9	14	2.6%
10	13	2.4%

The hypothesized explanation is that the first iteration is loaded from the flash memory, however, the subsequent iterations are loaded from the cache. Thus the timing is different since loading instructions from the flash memory is expected to take a longer time.

To investigate that, a simple program as shown in Code 5.2 was used to measure the execution time of 100 Compressed No Operation (C.NOP)s. The program asserts the EM system's trigger, executes 100 C.NOPs and finally de-asserts the trigger. The program iterates for 10 rounds to measure the difference between iterations. The same program was loaded once on the flash memory and once on the ITIM, which is supposed to be one way of the two-way set-associative cache as described in Section 4.2.

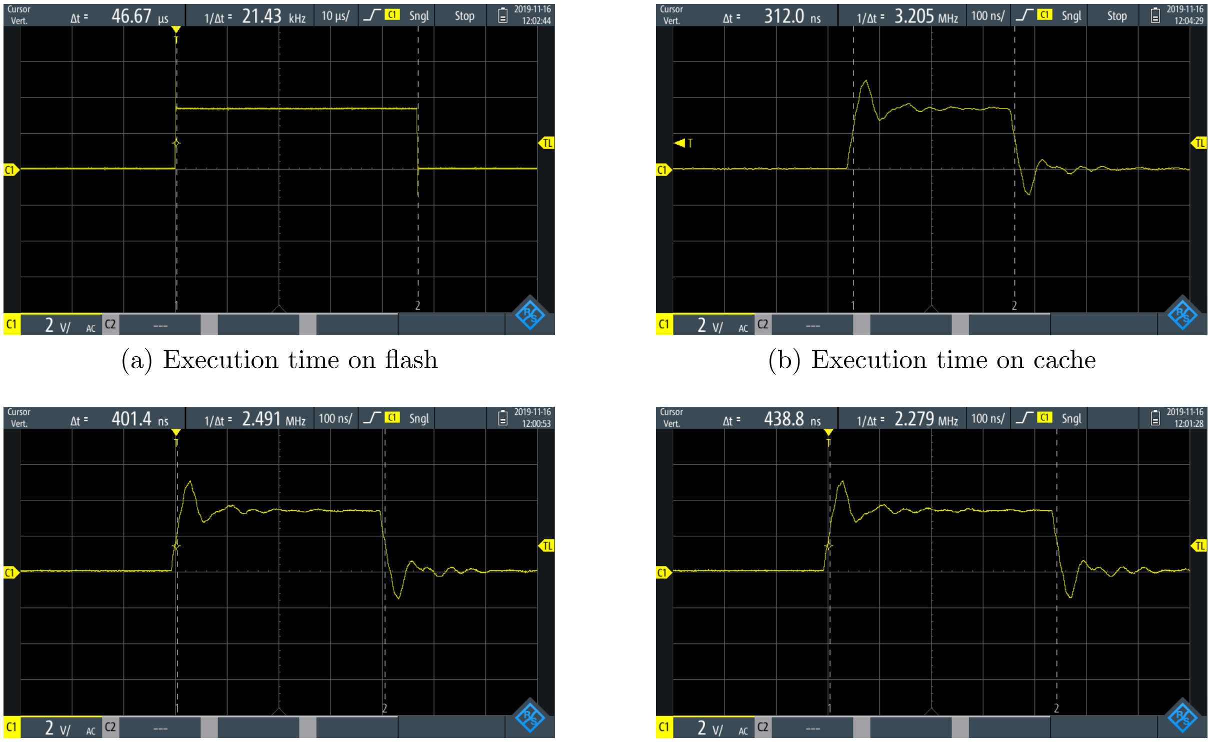
Code 5.2: Test program to check execution time of flash memory as well as cache and ITIM

```
main:

    #Initializations
    li x22,10                   #Number of rounds for the main loop 10

    #Main Loop
    LOOP:

        (Assert the trigger)    #Sets an IO to assert the trigger

        C.NOP
              .
              .
        (100 times in total)
              .
              .
        C.NOP

        (De-Assert the trigger) #Clears an IO to de-assert the trigger

        addi x22,x22,-1         #Decrements rounds counter by 1
        bne x22,x0,LOOP         #keeps in loop until counter is 0
```

Since the frequency used in the experiment is 320 MHz, which means a period of 3.125 ns per cycle, 100 cycles are expected to be executed in 312.5 ns on ITIM for the 10 iterations, a longer time on the flash memory for the first iteration but 312.5 ns for the 9 subsequent iterations since instructions are supposed to be cached by then.

The results obtained as shown in Figure 5.4 confirm that the execution time on flash memory took a long time (46.67 μs) for the first time, but the subsequent cached iterations took 312.5 ns. However, for the ITIM it showed a variant execution time around 400 ns for the first iteration and 312.5 ns for the subsequent iterations.

The long execution time of the first iteration on flash memory explains why no faults were injected on the first iteration. However, the unpredictable execution time of the ITIM for the first iteration makes it unreliable to use. Thus, all further experiments use the flash memory with codes using a loop of two iterations, relying on the second iteration only, knowing that the first iteration will never inject a fault.



(a) Execution time on flash

(b) Execution time on cache

(c) Execution time on ITIM

(d) Execution time on ITIM

Figure 5.4: Execution time of 100 NOPs using different memory configurations

5.3.2 Identifying the Best Position For the Target Instruction

The experiment in Section 5.1 identified the hottest spot to inject faults, but still, the exact timing on which instruction is faulted is not yet identified. While it is known that trigger-to-pulse delay of the EM system is around 290 ns as shown in Figure 4.4, which is 93 cycles if using a clock frequency of 320 MHz. But this delay is measured from the time the trigger signal reaches 50% of its peek, which actually consumes around an additional five ns. Also that delay does not account for the time consumed executing the store instruction responsible for asserting the trigger. Moreover, this store instruction is actually a store to the memory mapped GPIO, which has a larger execution time than a regular store to memory. Thus the real timing when the target instruction is faulted is definitely beyond 93 cycles.

In this experiment, the exact timing is measured empirically. The program in Code 5.3, puts N C.NOPs after the trigger assertion, followed by only one ADD instruction as a target, followed by 10 C.NOPs afterwards to ensure isolation of the target instruction. This experiment was repeated with 20 variation each with a different N numbers for C.NOPs ranging from 101 to 120. Each variation was run for 100 rounds for statistical analysis.

Code 5.3: Test program to determine the best position for the target instruction after trigger assertion

```
main:

    #Initializations
    li x22,2                    #Number of rounds for the main loop 2
    addi x19,x0,0x99            #Initializes Error Code to be 0x99

    #Main Loop
    LOOP:

        li x16,7                #Correct Sum
        li x17,5                #Initializes x17 to be 5
        li x13,2                #Initializes x13 to be 2

        (Assert the trigger)    #Sets an IO to assert the trigger

        C.NOP
              .
              .
        (N times in total)
              .
              .
        C.NOP

        add x17,x17,x13

        C.NOP
              .
              .
        (10 times in total)
              .
              .
        C.NOP

        sub x14,x16,x17         #x14=x16 - x17
        bne x14,x0,FAULT        #If x14 not 0, jump to FAULT

        (De-Assert the trigger) #Clears an IO to de-assert the trigger

        addi x22,x22,-1         #Decrements rounds counter by 1
        bne x22,x0,LOOP         #keeps in loop until counter is 0

    #No Fault Injected
    NO_FAULT:
        addi x19,x0,0xAA        #Error Code 0xAA for no fault
        j NO_FAULT

    #Fault Injected
    FAULT:
        addi x19,x0,0xBB        #Error Code 0xBB for fault injected
        j FAULT

    #Exception Fault
    early_trap_vector:
        csrr x5,mcause          #Exception code for the exception cause
        csrr x6,mepc            #PC when the exception took place
        csrr x7,mtval           #The corrupted instruction/address
        addi x19,x0,0xCC        #Error Code 0xCC for exception fault
        j early_trap_vector
```

The results of this experiment are shown in Figure 5.5 which represents a bar chart for the number and the type of faults injected per each target instruction position, ranging from 101 to 120 C.NOPs after trigger assertion. The results could be discussed as follows:

- It seems the exact positioning of the target instruction is not very strict, and faults can be injected to many positions. This could be justified by the jitter the EM system has as depicted in Figure 4.4, thus the pulse might hit different instruction positions.

- Positioning the target instruction after 109 C.NOPs gives the highest probability for fault injection. Which will be the position used for further experiments analyzing fault injection on individual instructions in Section 5.5. It can also be noticed that in terms of number of instructions skipped, positioning the target instruction after 110 C.NOPs would give a higher probability of instruction skips.

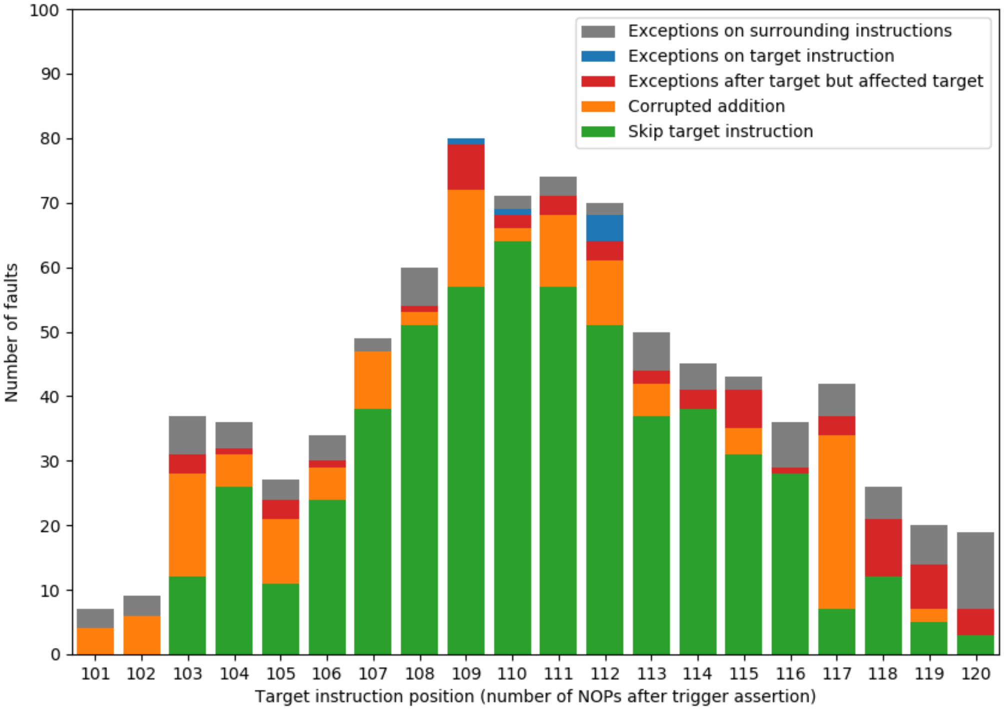


Figure 5.5: Timing analysis for target instruction positioning and corresponding types of faults injected. Pulse amplitude = 170 V, VDD = 1.8 V and frequency = 320 MHz

- Instructions are more likely to be skipped than producing corrupted outputs. These corrupted outputs appear as random numbers stored in the destination register instead of the correct sum.

- In rare cases target instructions might get corrupted itself resulting in exception faults being raised. With further investigation using the "mcause" register, the exception appears to happen due to the instruction being decoded as an illegal instruction. Looking at the "mtval" register, it shows that the corrupted instructions decoded were all 0s, which is defined as illegal instruction by the RISC-V ISA.

- Not only the target instructions are the ones to get corrupted, but also the surrounding C.NOPs tend to get corrupted as well due to the jitter of the EM system. But since a C.NOP is encoded as 0x0001, it is much easier to be corrupted to all 0s causing exception faults.

- More interestingly, it is noticed that in some cases, although an exception was raised, and with the knowledge of the "mepc" register, it is concluded that the exception occurred to a C.NOP placed after the target position. However, the destination register of the ADD instruction appears to contain corrupted data or having the initial value meaning the instruction was corrupted or skipped. Meaning that both the target instruction and that following C.NOP were faulted with one EM pulse. Which is the motivation behind the next experiment in Section 5.4.

5.4 Faulting Multiple Instructions

As was noticed from the experiment in 5.3, a skip to the target instruction as well as an exception fault to a following C.NOP was experienced, meaning that two instructions were faulted at once. Also, the existence of a relatively large window of time when an instruction can be faulted, as depicted in Figure 5.5, points to the possibility of having multiple instructions faulted at once.

To test that possibility, the program in Code 5.4 was used. The program targets 10 ADD instructions, all having the same two input operands but 10 different destination registers. The 10 ADD instructions are placed after 107 C.NOPs after the trigger. That was chosen based on Figure 5.5 which shows that the instructions after 107 to 116 C.NOPs after the trigger assertion are the most probable ones to be faulted. The 10 destination registers are initialized at the beginning of the program to the same initial value. The logic behind the experiment is that if no faults are injected all 10 registers should have the same ADD operation result, however, if some keep the initial value or a wrong result then their corresponding ADD instructions were skipped or corrupted.

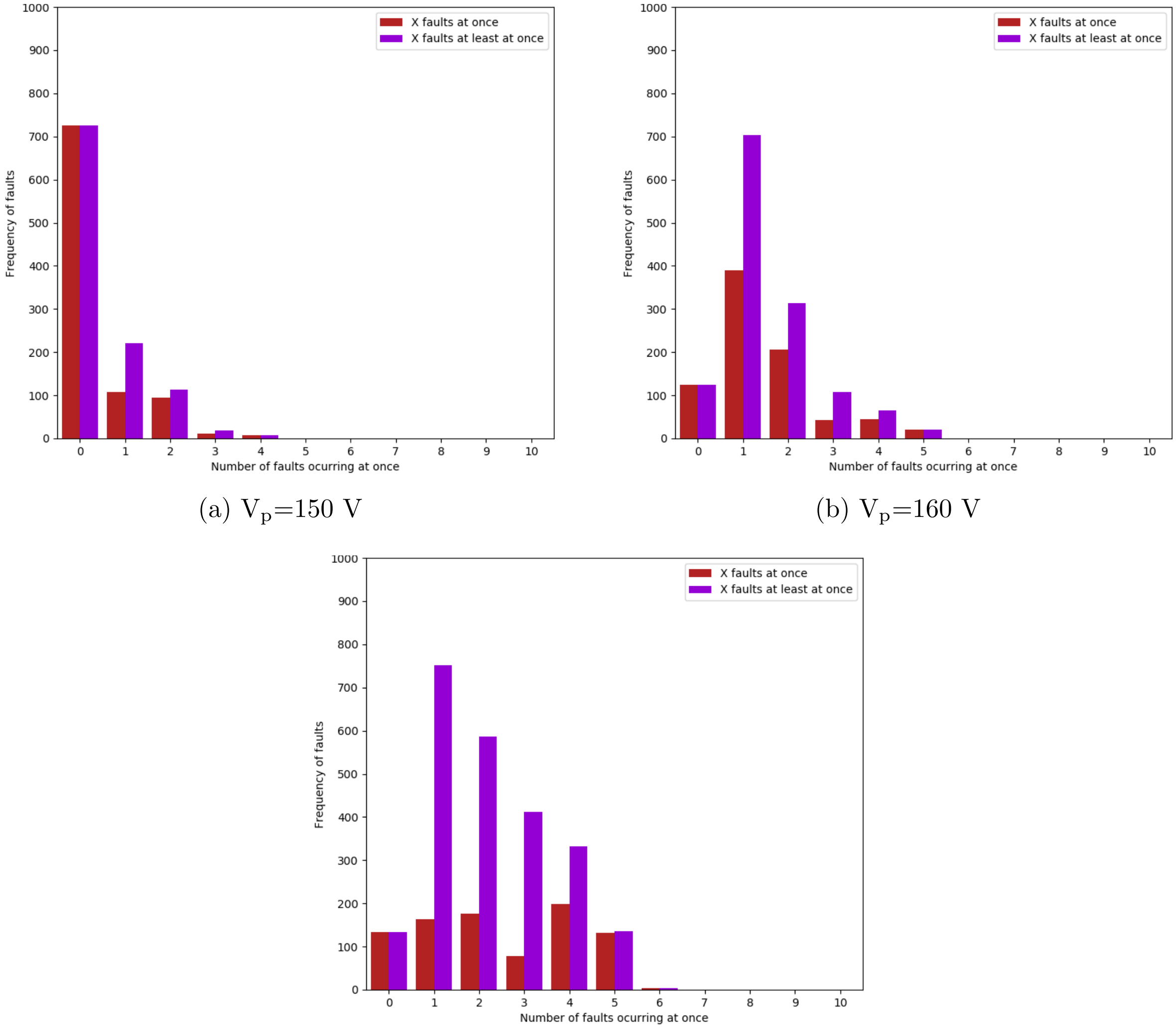
The experiment was performed with different EM pulse amplitudes and repeated 1000 times for each to perform statistical analysis. It is worth to mention that only instruction skips were included in the analysis. Exception faults were excluded from this analysis as if an exception occurs it would prevent the following instructions from being faulted anyway. Also, Corrupted results were not included being rare and also corrupted results make the fault detectable if instruction duplication is used by the developer. Thus, multiple skips are more important to analyze.

Code 5.4: Test program to explore if multiple instructions can be faulted at once

```
main:

    #Initializations
    li x22,2                    #Number of rounds for the main loop 2
    addi x19,x0,0x99            #Initializes Error Code to be 0x99

    #Main Loop
    LOOP:

        li x12,3                #Initializes first operand x12 to be 3
        li x13,4                #Initializes second operand x13 to be 4

        li x14,5                #Initializes destination reg x14 to be 5
        li x15,5                #Initializes destination reg x15 to be 5
        li x16,5                #Initializes destination reg x16 to be 5
        li x17,5                #Initializes destination reg x17 to be 5
        li x18,5                #Initializes destination reg x18 to be 5
        li x24,5                #Initializes destination reg x24 to be 5
        li x28,5                #Initializes destination reg x28 to be 5
        li x29,5                #Initializes destination reg x29 to be 5
        li x30,5                #Initializes destination reg x30 to be 5
        li x31,5                #Initializes destination reg x31 to be 5

        (Assert the trigger)    #Sets an IO to assert the trigger

        C.NOP
              .
              .
        (107 times in total)
              .
              .
        C.NOP

        add x14,x12,x13         #1st target instruction
        add x15,x12,x13         #2nd target instruction
        add x16,x12,x13         #3rd target instruction
        add x17,x12,x13         #4th target instruction
        add x18,x12,x13         #5th target instruction
        add x24,x12,x13         #6th target instruction
        add x28,x12,x13         #7th target instruction
        add x29,x12,x13         #8th target instruction
        add x30,x12,x13         #9th target instruction
        add x31,x12,x13         #10th target instruction

        C.NOP
              .
              .
        (10 times in total)
              .
              .
        C.NOP

        (De-Assert the trigger) #Clears an IO to de-assert the trigger

        addi x22,x22,-1         #Decrements rounds counter by 1
        bne x22,x0,LOOP         #keeps in loop until counter is 0

    #No Exception Faults
    NO_EXCEPTION:
        addi x19,x0,0xAA        #Error Code 0xAA for no exception
        j NO_EXCEPTION

    #Exception Fault
    early_trap_vector:
        csrr x5,mcause          #Exception code for the exception cause
        csrr x6,mepc            #PC when the exception took place
        csrr x7,mtval           #The corrupted instruction/address
        addi x19,x0,0xCC        #Error Code 0xCC for exception fault
        j early_trap_vector
```

The empirical EMFI results indicate that multiple instructions can be faulted at once. Figure 5.6 show the number of instructions skipped at once with different EM pulse amplitudes (150 V, 160 V and 170 V). As a general observation, not only the number of faults increases by the increase of the EM pulse amplitude which is observed in previous experiments, but also the number of instructions faulted at once increases.

Figure 5.6a shows that with EM pulse amplitude of 150 V, the maximum number of instructions skipped at once is four. Even though, it is unlikely to have three or four instructions skipped at once. However, the probability of having two instructions skipped at once is non-negligible being 9.4% and similar to having only one instruction skipped, which has a probability of 10.8%. In total, the probability of having no skips at all is as high as 72.5%, while the probability of having at least one instruction skipped is 22.1%, in addition to 5.4% of the rounds having exception faults.



(a) V_p=150 V

(b) V_p=160 V

(c) V_p=170 V

Figure 5.6: Histogram for the number of ADD instructions skipped at once. VDD = 1.8 V and frequency = 320 MHz

In Figure 5.6b a dramatic fall in the probability of having no skips is observed as it reached 12.4% only when the EM pulse amplitude is increased to 160 V. In contrast, the probability of having at least one instruction skipped dramatically increased to reach 70.3%, in addition to 17.3% of the rounds having exception faults. The probability of having multiple instructions skipped at once increased notably especially for two instructions being skipped at once reaching a probability of 20.6%. Even with a small probability having five instructions skipped at once became possible.

While with an EM pulse amplitude of 170 V, as displayed in Figure 5.6c the probability of having no skips at all did not change much compared to when the EM pulse amplitude was 160 V. However, instead of having the highest probability towards having only one instruction skipped, the probability of having skips is distributed over the different possibilities of having two, three, four and five instructions skipped at once being 17.6%, 7.8%, 19.8%, and 13.1% respectively. Moreover, a low probability for having six instructions skipped emerged. With these results, the probability of having at least four instructions skipped at once becomes 33.3%, the probability of having at least three instructions skipped at once is 41.4% and for having at least two instructions skipped at once the probability goes as high as 58.7%. Meaning that with an EM pulse amplitude of 170 V, having multiple instructions skipped at once is more probable than having only one instruction skipped or no instructions skipped at all.

While the target processor is treated as a black box, with not much knowledge of the actual implementation in silicon, one cannot determine for sure the effect of the EM pulse on the core. However, possible explanations for having multiple instructions being skipped or faulted at once could be portrayed as follows:

- The long rise time of the generated EM pulse as displayed previously in Figure 4.4, which ranges from seven ns to 12 ns can roughly cover four instructions working with 320 MHz clock frequency. Thus, could be responsible for the multiple instructions skips.
- The fact that the target is a five-stage pipelined processor, and the ability of the EM pulse to have a global effect, makes it possible to affect up to five different instructions in the different five pipeline stages.
- By combining the two previous hypotheses, assuming that five instructions in the five pipeline stages are affected, in addition to the effect of the rise time that can cover four instructions, which effectively can add only three more instructions, then up to eight instructions can be hypothetically be faulted at once.

5.5 Analysis of Faulted Instructions

In this section, experiments on the individual instructions are presented to analyze the effect of EMFI on different instructions. The RISC-V base instructions could be categorized into three main categories:

- **Arithmetic and Logic Operations** which are mainly performed by the Arithmetic Logic Unit (ALU) such as addition, bit-wise ANDing, ORing, etc.
- **Memory Operations** used to move data from memory to registers and vice versa.
- **Flow Control Operations** which accommodates procedure calls and conditional execution, such as jump and branch instructions.

For each category, a set of instructions was chosen to perform the analysis. The programs used for experiments follow the template in Code 5.5 that fixes the number of C.NOPs prior to the target to be 109 C.NOPs as it provides the highest probability of fault injection as shown in Figure 5.5. All experiments were performed using a clock frequency of 320 MHz with a supplied voltage of 1.8 V and EM pulse amplitude of 170 V. For each instruction studied, the experiments were repeated 1000 times to perform statistical analysis.

Code 5.5: Template program to examine individual instructions fault injection

```
main:

    #Initializations
    li x22,2                    #Number of rounds for the main loop 2
    addi x19,x0,0x99            #Initializes Error Code to be 0x99

    #Main Loop
    LOOP:

        ###Registers Initialization###

        (Assert the trigger)    #Sets an IO to assert the trigger

        C.NOP
              .
```

```
          .
    (109 times in total)
          .
          .
    C.NOP

    ###Target Instruction###

    C.NOP
          .
          .
    (10 times in total)
          .
          .
    C.NOP

    ###Check Result###

    (De-Assert the trigger) #Clears an IO to de-assert the trigger

    addi x22,x22,-1         #Decrements rounds counter by 1
    bne x22,x0,LOOP         #keeps in loop until counter is 0

#No Fault Injected
NO_FAULT:
    addi x19,x0,0xAA        #Error Code 0xAA for no fault
    j NO_FAULT

#Fault Injected
FAULT:
    addi x19,x0,0xBB        #Error Code 0xBB for fault injected
    j FAULT

#Exception Fault
early_trap_vector:
    csrr x5,mcause          #Exception code for the exception cause
    csrr x6,mepc            #PC when the exception took place
    csrr x7,mtval           #The corrupted instruction/address
    addi x19,x0,0xCC        #Error Code 0xCC for exception fault
    j early_trap_vector
```

5.5.1 Arithmetic and Logic Operations

The instructions examined under this category are: ADD, SUB, AND, OR, and XOR. The instructions formats and descriptions are presented in Table 5.2

Table 5.2: Instructions formats and descriptions for arithmetic and logic operations

Instruction	Description
ADD rd, rs1, rs2	Adds rs1 to rs2 and stores the result to rd
SUB rd, rs1, rs2	Subtracts rs2 from rs1 and stores the result to rd
AND rd, rs1, rs2	Performs ANDing on rs1 and rs2 and stores the result to rd
OR rd, rs1, rs2	Performs ORing on rs1 and rs2 and stores the result to rd
XOR rd, rs1, rs2	Performs XORing on rs1 and rs2 and stores the result to rd

The programs used to examine the arithmetic and logic operations follow the template in Code 5.5. The modifiable part in the template is filled as shown in Code 5.6 for the OR instruction.

Code 5.6: The modified part of Code 5.5 to examine OR instruction fault injection

```
LOOP:

    ###Registers Initialization###
    li x12,0x331F99E5   #Initializes first  operand  x12
    li x13,0xF1EBCEAD   #Initializes second  operand  x13
    li x16,0xF3FFDFED   #Initializes x16 to the correct result
    li x17,0xAAAAAAAA   #Initializes reg x17 with 0xAAAAAAAA
              .
              .
    ###Target Instruction###
    or x17,x13,x12
              .
              .
    ###Check Result###
    sub x14,x16,x17     #x14 = x16 - x17
    bne x14,x0,FAULT    #If x14 not 0, jump to FAULT
```

The program in Code 5.6 initializes the registers x12 and x13 with two random numbers, initializes x16 with the correct result of the OR operation on both x12 and x13. The target instruction utilizes the OR instruction using x12 and x13 as operands and x17 as a destination registers. Eventually, the result of the target instruction in x17 is compared with the expected result stored in x16 to determine if a fault is injected or not. The same program in Code 5.6 is utilized for other arithmetic and logic operations with changes according to the instruction under test.

The results in Table 5.3 show a similar effect on all examined arithmetic and logic operations. On average, around 61% of the rounds were performed correctly without any faults. The majority of faults were instruction skips, representing 29% of rounds on average. However, some corrupted results were reported instead of instruction skips, representing an average of 9%. Also, on average, around 1% of the rounds examined exception faults on the surrounding C.NOPs.

Table 5.3: Faults analysis for arithmetic and logic operations

Fault Type	Instruction				
	ADD	SUB	AND	OR	XOR
No fault	64.6%	59.1%	60.4%	59.2%	62.2%
Instruction skip	28.3%	27.2%	32.9%	30.1%	26.2%
Corrupted result	5.0%	10.9%	6.3%	10.4%	11.6%
Exception fault	2.1%	2.8%	0.4%	0.3%	0%

5.5.2 Memory Operations

Only LW (Load Word) and SW (Store Word) were analyzed under this category. Half word and byte memory operations were not considered since they are not widely used and are expected to give the same results. The instructions formats and descriptions are presented in Table 5.4

The programs used to examine the LW and SW instruction follow the template in Code 5.5, with the modifiable part filled with the Code 5.7 for the LW instruction and similarly in Code 5.8 for the SW instruction.

Table 5.4: Instructions formats and descriptions for memory operations

Instruction	Description
LW rd, imm(rs1)	1- Adds rs1 register to imm value to form the address x. 2- Loads data from memory with address x to the register rd.
SW rs2, imm(rs1)	1- Adds rs1 register to imm value to form the address x. 2- Stores data from the register rs2 to memory with address x.

The program in Code 5.7 stores pre-determined data to the memory using the address 0x80001000. The target instruction loads the data stored in the same address to the register x17. Eventually, the data in x17 is compared with the expected data which was stored previously in memory.

Code 5.7: The modified part of Code 5.5 to examine LW instruction fault injection

```
LOOP:

    ###Registers Initialization###
    li x13,0x80001000   #Initializes x13 with the memory address
    li x16,0xF1EBCEAD   #Initializes x16 with the data to be stored
    li x17,0xAAAAAAAA   #Initializes reg x17 with 0xAAAAAAAA
    sw x16,0(x13)       #Stores data from x16 to memory
              .
              .
    ###Target Instruction###
    lw x17,0(x13)       #Loads data from memory to reg x17
              .
              .
    ###Check Result###
    sub x14,x16,x17     #x14 = x16 - x17
    bne x14,x0,FAULT    #If x14 not 0, jump to FAULT
```

The program in Code 5.8 stores initial data to the memory using the address 0x80001000. The target instruction overwrites that initial data and stores pre-determined data to the same memory address. Eventually, the data in that memory address is loaded to x17 and compared with the data which was expected to be stored by the target instruction.

Code 5.8: The modified part of Code 5.5 to examine SW instruction fault injection

```
LOOP:

    ###Registers Initialization###
    li x13,0x80001000   #Initializes x13 with the memory address
    li x16,0xF1EBCEAD   #Initializes x16 with the data to be stored
    li x17,0xAAAAAAAA   #Initializes reg x17 with 0xAAAAAAAA
    sw x17,0(x13)       #Initialize the data in memory with 0xAAAAAAAA
              .
              .
    ###Target Instruction###
    sw x16,0(x13)       #Stores data from reg x17 to memory
              .
              .
    ###Check Result###
    lw x17,0(x13)       #Loads data stored in memory to reg x17
    sub x14,x16,x17     #x14 = x16 - x17
    bne x14,x0,FAULT    #If x14 not 0, jump to FAULT
```

The results displayed in Table 5.5 show that for the LW instruction, 43.6% of the rounds were performed correctly with no faults. The probability of having an LW instruction skipped is 33.8%, while with a probability of 4.3% the LW instruction is performed with corrupted data being loaded to the destination register. Also, exception faults to the surrounding C.NOPs appeared in 18.3% of the rounds.

For the SW instruction, 52% of the rounds did not experience any faults. Interestingly, throughout the 1000 rounds, none of the rounds experienced corrupted data being stored in memory, however, 32.2% experienced instruction skip. Similarly, with a probability of 15.8%, the surrounding C.NOPs were reported to experience exception faults.

Table 5.5: Faults analysis for memory operations

Fault Type	Instruction	
	LW	SW
No fault	43.6%	52.0%
Instruction skip	33.8%	32.2%
Corrupted result	4.3%	0%
Exception fault	18.3%	15.8%

5.5.3 Flow Control Operations

For the flow control operations, the following instructions were analyzed: JAL (Jump and Link), JALR (Jump and Link Register) and BNE (Branch Not Equal). The instructions formats and descriptions are presented in Table 5.6.

Table 5.6: Instructions formats and descriptions for control flow operations

Instruction	Description
JAL rd, imm	1- Constructs address x by shifting imm value left by 1. 2- Jumps to the instruction with address x. 3- Adds 4 to the program counter and stores the result in rd.
JALR rd, imm(rs1)	1- Constructs address x by adding rs1 register to imm value. 2- Jumps to the instruction with address x. 3- Adds 4 to the program counter and stores the result in rd.
BNE rs1, rs2, imm	1- Constructs address x by shifting imm value left by 1 and adding the result to the program counter. 2- Jumps to the instruction with address x if rs1 $\neq$ rs2. Otherwise the program counter increments normally.

The programs used to examine the JAL, JALR and BNE instructions follow the template in Code 5.5, with the modifiable part filled with Code 5.9 for JAL and JALR instructions, and similarly in Code 5.10 for BNE instruction

The program in Code 5.9 initializes x14 to be 2, then the target instruction JAL jumps to a label named BRANCHED and stores the address of the next instruction (Program Counter (PC) + 4) to x13. IF the target instruction is executed correctly then x14 will be assigned the value 5, then the program will jump to the CHECK label. If the target instruction is skipped then x14 will maintain the value 2 and the program will proceed to the CHECK label anyway. Eventually, x14 is checked to determine if the jump was executed or not, also x13 is compared with the expected linked PC. The program used to test JALR instruction is similar to the one in Code 5.9, only it uses JALR as a target instruction (jalr x13,x17), with the label's address pre-loaded to x17.

Code 5.9: The modified part of Code 5.5 to examine JAL instruction fault injection

```
LOOP:
    ###Registers Initialization###
    li x12,0x200102ac       #Initializes x12 with the Correct linked PC
    li x13,0xAAAAAAAA       #Initializes x13 with 0xAAAAAAAA
    li x14,2                #Initializes x14 with 2
    li x16,5                #Initializes x16 with 5
              .
              .
    ###Target Instruction###
    jal x13,BRANCHED        #Jumps to BRANCHED and stores (PC + 4) to x13
              .
              .
    ###Check Result###
    CHECK:
        bne x14,x16,FAULT   #If didn't jump to BRANCHED, jumps to FAULT
        bne x12,x13,FAULT   #If linked PC is not correct, jumps to FAULT
              .
              .
    BRANCHED
        li x14,5            #Loads x14 with 5
        j CHECK             #Jump to CHECK
```

The results in Table 5.7 show that jump instructions have never been skipped. However, the address saved in the linking process is corrupted in 29.8% and 26% of the rounds for JAL and JALR respectively. On average, 56% of the jumps were executed correctly, and 15.5% of the rounds experienced exception faults on the surrounding C.NOPs.

Table 5.7: Faults analysis for unconditional jump operations

Fault Type	Instruction	
	JAL	JALR
No fault	53.5%	58.8%
Instruction skip	0%	0%
Corrupted link	29.8%	26.0
Exception fault	16.4%	14.7%

Since BNE is a conditional branch instruction, two different experiments were conducted on the instruction, once with a positive condition where the branch should be taken, and another with a negative condition where the branch should not be taken. The program in Code 5.10 examines BNE instruction in a positive condition. The code initializes x12 and x13 with two different values and initializes x14 with 2. The target instruction compares x12 and x13 and jumps if they are not equal. Since they are not equal, if the instruction was executed without faults it should jump to BRANCHED label where x14 would be loaded with 5 and eventually jumps to CHECK label to determine if the branch was taken or not. The program used to examine BNE in a negative scenario is similar to the one in Code 5.10 except for initializing both x12 and x13 with the same value.

Code 5.10: The modified part of Code 5.5 to examine BNE instruction fault injection

```
LOOP:

    ###Registers Initialization###
    li x12,0x331F99E5       #Initializes x12 with a random value
    li x13,0xF1EBCEAD       #Initializes x13 with a random value
    li x14,2                #Initializes x14 with 2
    li x16,5                #Initializes x16 with 5
              .
              .
    ###Target Instruction###
    bne x12,x13,BRANCHED    #Jumps to BRANCHED if x12 is not equal to x13
              .
              .
    ###Check Result###
    CHECK:
        bne x14,x16,FAULT   #If didn't jump to BRANCHED, jumps to FAULT
              .
              .
    BRANCHED
        li x14,5            #Loads x14 with 5
        j CHECK             #Jump to CHECK
```

The results depicted in Table 5.8 show that for the BNE instruction with a positive condition, 96.1% of the rounds were performed correctly with no instruction skips. The only faults occurred were the exception faults on the surrounding C.NOPs. For the negative scenario where the branch is not expected to be taken, it is infeasible to know if the instruction is skipped or executed correctly since both would result in the branch not to be taken. These two possibilities combined represent 80% of the rounds. More interestingly, 19.1% of the rounds took the branch when they were not expected to. Exception faults on the surrounding C.NOPs were present on 0.9% of the rounds.

Table 5.8: Faults analysis for conditional branch operations

Fault Type	Instruction	
	BNE +VE	BNE -VE
No fault	96.1%	80%
Instruction skip	0%	NA%
False jump	NA%	19.1%
Exception fault	3.9%	0.9%

Chapter 6

XOR BR/TBR PUFs Design on FPGA

As discussed in Section 3.6, attacks on single BR and TBR PUF implementations on FPGA were successful in previous research. However, modelling attacks in the literature were not successful on XOR BR PUFs with more than three instances XORed. Hence, the motivation to implement 4-input XOR BR an TBR PUFs with good characteristics on FPGAs to serve as a base for launching more advanced modelling attacks and investigations, such as in [33] [34] where deep learning techniques were applied on these implementations. Though, in this thesis the focus is only on the hardware implementation aspects, while the attack techniques are detailed in [33] [34].

This chapter starts by a system level overview of the hardware and software components used to implement and evaluate the PUFs in Section 6.1. The evaluation metrics for the implemented PUFs are explained in detail in Section 6.2 before discussing the realization of both BR and TBR PUFs on FPGAs and their mappings into Look Up Table (LUT)s in Section 6.3. Section 6.4 discusses two different approaches on implementing single BR PUFs on FPGA, namely the automatic and manual placement approaches. And finally, Section 6.5 and Section 6.6 discuss the implemented XOR BR and TBR PUFs respectively and their proposed optimizations. As well as listing their characteristics and their performance against modelling attacks according to [33] [34].

6.1 Overall System Architecture

For the hardware implementations of PUFs, Mojo V3 boards were used. The board in Figure 6.1 features a Spartan-6 XC6SLX9 Xilinx FPGA with a clock source of 50 MHz, alongside with an ATmega32U4 AVR microcontroller with an 8 MHz clock crystal.

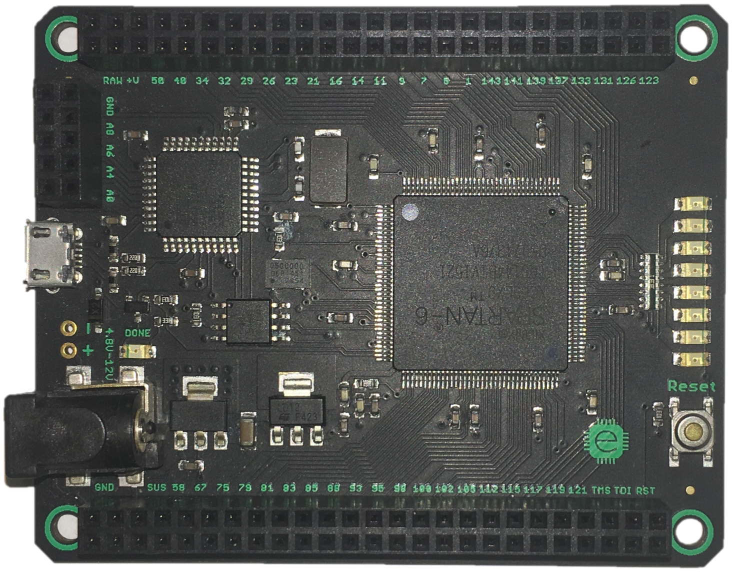


Figure 6.1: Mojo Board

As shown in Figure 6.2, the PUF is implemented on the FPGA side, in addition to a Finite State Machine (FSM) for control, as well as a Universal Asynchronous Receiver-Transmitter (UART) module for receiving challenges and sending responses. The UART module on top of the FPGA is connected to the UART of the AVR microcontroller, which in return transfers the data through the USB port of the Mojo board connected to an external PC.

A script employed on the PC side is responsible for generating and sending random challenges to the board, receiving the responses and storing them into log files in order to evaluate the PUF's characteristics as will be thoroughly discussed in Section 6.2.

Challenges are generated using a Galois Linear-Feedback Shift Register (LFSR) pseudo-number generator, which as displayed in Figure 6.3 is basically a shift register, however, some bits (also known as taps) are XORed with the output bit before being shifted. To obtain maximum length LFSRs for different PUF sizes, the taps in Table 6.1 were used.

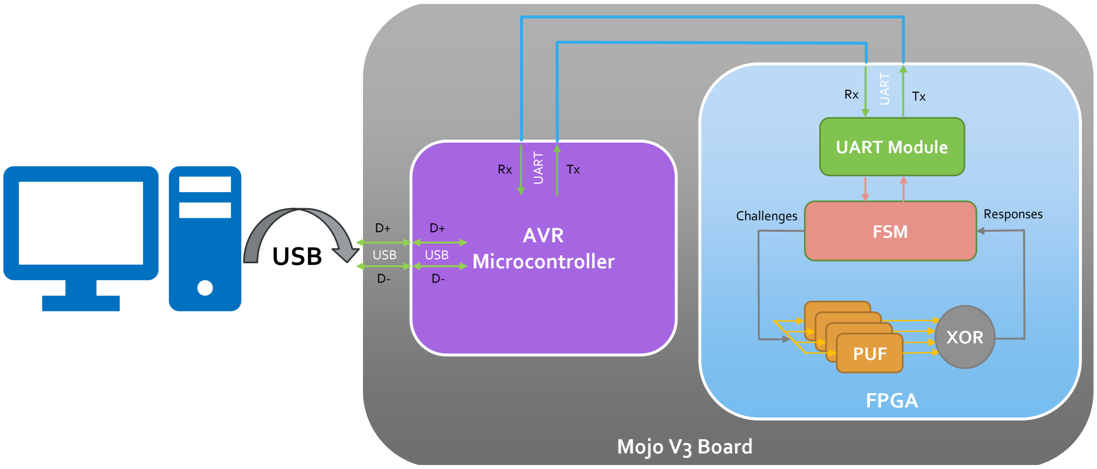


Figure 6.2: PUF Ecosystem

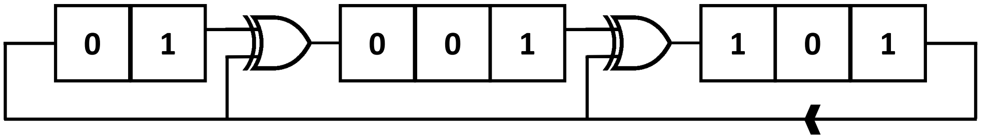


Figure 6.3: Galois Linear-Feedback Shift Register

Challenges are sent byte by byte through serial communication to the FPGA, where the FSM receives each byte, stores them and concatenates them to form the full challenge vector sent to the PUF. To make sure that the challenge is received correctly, the challenge bytes are XORed together forming one byte which is sent to the PC side to be compared with a similar byte generated from the transmitted challenge.

After the FSM forms the full challenge vector, it transfers it to the PUF at once as well as releases the reset signal for the PUF at the same time. Then the FSM waits for a predetermined time so that the PUF converges to a stable state before capturing the response. BR and TBR PUFs' response is usually the output of one stage of the ring, however, in this implementation, all the stages' outputs are derived to make sure that all stages have the same capacitive load, as otherwise one stage would have different load than others, which might bias the PUF. The same note was considered in the literature for the BR PUF as in [76]. Another advantage of deriving all stages' outputs is to distinguish

Table 6.1: LFSR taps for different PUF sizes

PUF Size	Taps Positions
64	64, 63, 61, 60 [3] [71]
128	128, 126, 101, 99 [3]
256	256, 254, 251, 246 [71]

between the stabilized and unstabilized responses. To do so, the response bytes are ORed together forming one byte that is sent to the PC side. For a stabilized ring, that byte should be either '10101010' (0xAA) representing '1' or '01010101' (0x55) representing '0', other than that, it indicates an unstabilized ring, which should not be added to the CRPs database.

6.2 PUFs Evaluation Metrics

As previously discussed in Section 3.5, a strong PUF should achieve three main properties: reproducibility, unpredictability and uniqueness [53] [12]. In this section, the evaluation metrics that measure these three properties are defined. These metrics are: PUF noise, PUF bias, and Inter-Chip hamming distance. Also, the individual challenge bits influence is considered being another important characteristic of PUFs.

- **PUF Noise**: A reproducible PUF would give a consistent response to a certain challenge per each chip, however, in reality, a PUF might give inconsistent responses for the same challenge. To measure noise, the same challenge is applied to the same chip for many iterations, a majority vote is taken to determine the supposedly right response and the same process is repeated for all challenges. Then, the noise can be calculated as in Equation 6.1 as the number of deviating responses over the number of iterations applied, taking the average eventually over all challenges applied.

$$N = \frac{\sum \# \; wrong\ responses}{\# \; iterations \times \# \; challenges} \tag{6.1}$$

  A reliable PUF should have an ideal noise of 0.

- **PUF Bias**: PUF bias represents the tendency of the PUF to respond with 0 or 1 more likely to different challenges. Bias can be calculated as the number of responses representing '1' over the total number of challenges applied as in Equation 6.2.

  $$B = \frac{\#\ responses\ of\ '1'}{\#\ challenges} \tag{6.2}$$

  An unpredictable PUF should have an ideal bias of 0.5.

- **Individual Challenge Bits Influence**: Challenge bits should ideally contribute equally to the resulted response and not by only some of the challenge bits, which affects the unpredictability of the PUF. For each challenge bit, its influence is calculated as the number of responses representing '1' over the total number of challenges with that bit being '1' as in Equation 6.3, also the influence is calculated when that challenge bit is '0' as in Equation 6.4.

  $$Infl(i,1) = \frac{\#\ responses\ of\ '1'}{\#\ challenges\ with\ ith\ bit = 1} \tag{6.3}$$

  $$Infl(i,0) = \frac{\#\ responses\ of\ '1'}{\#\ challenges\ with\ ith\ bit = 0} \tag{6.4}$$

  An unpredictable PUF should have an ideal influence of each bit as 0.5.

- **Inter-Chip Hamming Distance**: Different chips should give different responses to the same challenge. Inter-chip hamming distance represents how many responses were dissimilar for the same challenge on different chips. The normalized hamming distance between two different chips would be calculated as in Equation 6.3 as the number of dissimilar responses for the same challenge over the number of challenges.

  $$NHD = \frac{\#\ dissimilar\ responses}{\#\ challenges} \tag{6.5}$$

  A unique PUF should have an ideal normalized inter-chip hamming distance of 0.5.

In this research, to obtain the actual characteristics of the implemented PUFs, three typical Mojo boards were used. The same PUF designs were loaded on the three boards and 1 Million different challenges were applied on each for three iterations. All experiments were conducted at room temperature.

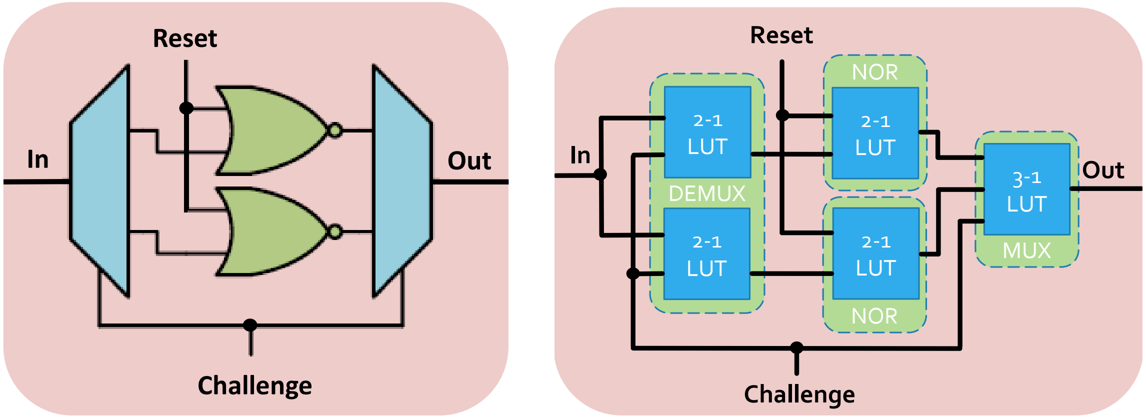
An important note is that as stabilized and unstabilized responses could be distinguished in the presented implementations, the unstabilized responses were excluded from the characteristics calculations.

6.3 BR/TBR PUFs Realizations on FPGA

This section, illustrates the realization of both BR and TBR PUFs and their mapping from schematic into FPGA LUTs. These mappings are the output of Xilinx ISE tool.

6.3.1 BR PUF Realization on FPGA

A single stage of the BR PUF as introduced in [14] should contain one MUX, one DEMUX and two inverting elements such as NOR gates as displayed in Figure 6.4a. Writing a Hardware Description Language (HDL) code describing such a circuit and trying to synthesize it for FPGA using Computer Aided Design (CAD) tools would result in optimizing it to only one LUT that has the functionality of one NOR gate. Although that is true from the digital logic point of view, it is not from the PUF functionality point of view. However, with the careful use of the CAD tool, one can control the optimizations and manage to separate the four components into five LUTs as shown in Figure 6.4b.



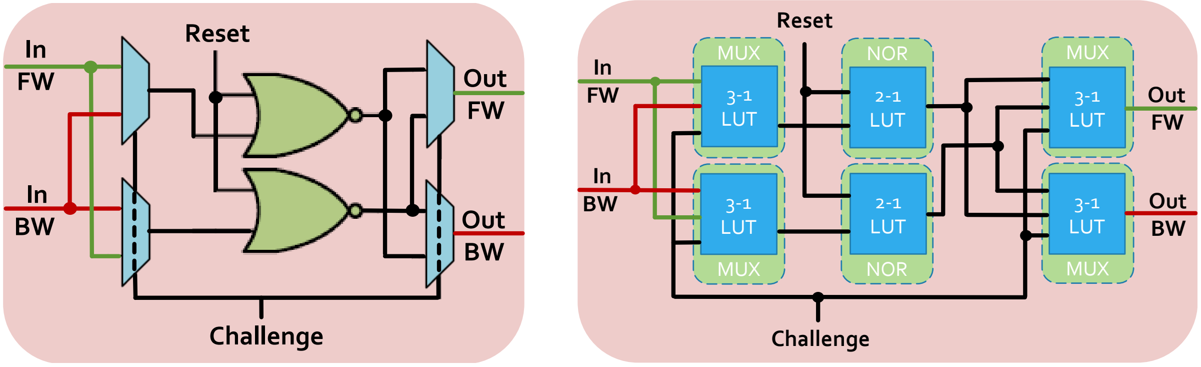
(a) Single BR PUF Stage Schematic [14]

(b) Single BR PUF Stage Layout

Figure 6.4: BR PUF Schematic and Layout on FPGA

6.3.2 TBR PUF Realization on FPGA

For the TBR PUF, the components of a single stage were not clarified in [58] but abstracted as displayed in Figure 3.6. However, it is reasonable to interpret that abstract design to the more detailed one in Figure 6.5a, which was also adopted in [25]. A straight forward implementation would try to transform each component of the PUF into a separate LUT as shown in Figure 6.5b, resulting in a total of 6 LUTs for one TBR PUF stage.



(a) Single TBR PUF Stage Schematic

(b) Single TBR PUF Stage Layout

Figure 6.5: TBR PUF Schematic and Layout on FPGA

6.4 Implementation Approaches

The FPGA development flow passes through six main processes: **circuit design** usually by writing HDL codes, **synthesising** HDL into netlists, **mapping** these netlists to the available FPGA resources, **placement** of these mappings onto the FPGA, **routing** and connecting these resources on FPGA and finally **generating the bitstream** to be loaded to the FPGA.

Xilinx ISE, the tool used for the development process, is capable of performing all the aforementioned processes automatically given the HDL code. Also, the option of manual placement and routing by the developer is available to allow for targeting specific performance improvements.

In this section, both automatic and manual placement approaches for BR PUF implementation are discussed with the corresponding implementations' characteristics presented.

6.4.1 Automatic Placement Approach

The automatic placement and routing approach is usually the most used approach used in FPGA development, unless further requirements need manual intervention. This approach was tested for the BR PUF implementation using five different settling times after which the PUF responses are taken. The main goal was to observe the responses convergence over time to check the point where waiting for more time would not make a difference.

Surprisingly, the results in Table 6.2 show that, for the three chips used for evaluation, the percentage of stabilized responses does not necessarily increase with the increase of the settling time, and it appears to be random. Moreover, the same chip tends to be biased towards 0 responses at some settling time, and with a different settling time it tends to be biased toward responses of 1. Hence, it indicates that with the change of the settling time, the whole placement and routing by the tool changes producing totally different PUFs.

Table 6.2: Automatic placement approach PUF characteristics

Metric	Settling Time (cycles)				
	64	128	256	512	1024
stabilized responses Chip 1 (%)	83	68	90	92	88
stabilized responses Chip 2 (%)	86	78	88	92	88
stabilized responses Chip 3 (%)	76	73	88	92	87
Bias Chip 1 (%)	12	55	17	60	41
Bias Chip 2 (%)	5	82	22	65	31
Bias Chip 3 (%)	21	71	31	36	53
Noise Chip 1 (%)	0.7	1.1	0.3	0.5	0.5
Noise Chip 2 (%)	1	0.7	0.4	0.4	0.4
Noise Chip 3 (%)	1.6	1	0.4	0.4	0.6
NHD Chips 1&2 (%)	21	44	20	20	25
NHD Chips 1&3 (%)	26	38	25	31	28
NHD Chips 2&3 (%)	30	35	23	36	33
NHD Average (%)	26	39	23	29	29
Max influence Chip 1 (%)	4	72	8	77	26
Max influence Chip 2 (%)	2	93	14	77	17
Max influence Chip 3 (%)	7	85	18	22	72

With this conclusion, another approach was suggested by using the settling time as a randomization source to produce different PUF implementations. And with the trial of many different settling times, some of the implementations would yield good PUFs with good properties.

The results shown in Table 6.3 are for some of the selected trials with much-improved properties. For the three presented settling times (500, 575 and 600 cycles), the percentage of the stabilized responses is around 90% which is good for BR PUFs. The bias is around the ideal value of 50% with an offset ranging from 3% to 16% which is acceptable. However, the inter-chip hamming distance between the chips is not very good especially with the cycles and 600 cycles settling time implementations. Also, it is very obvious that some individual challenge bits have a huge influence on the responses.

Table 6.3: Automatic placement approach PUF characteristics with selected settling times

Metric	Settling Time (cycles)		
	500	575	600
stabilized responses Chip 1 (%)	92	89	89
stabilized responses Chip 2 (%)	94	86	89
stabilized responses Chip 3 (%)	92	90	90
Bias Chip 1 (%)	64	39	57
Bias Chip 2 (%)	61	44	66
Bias Chip 3 (%)	47	59	60
Noise Chip 1 (%)	0.3	0.5	0.4
Noise Chip 2 (%)	0.3	0.6	0.5
Noise Chip 3 (%)	0.3	0.4	0.5
NHD Chips 1&2 (%)	20	30	26
NHD Chips 1&3 (%)	26	34	23
NHD Chips 2&3 (%)	24	31	25
NHD Average (%)	23	32	25
Max influence Chip 1 (%)	77	21	74
Max influence Chip 2 (%)	76	29	80
Max influence Chip 3 (%)	38	72	74

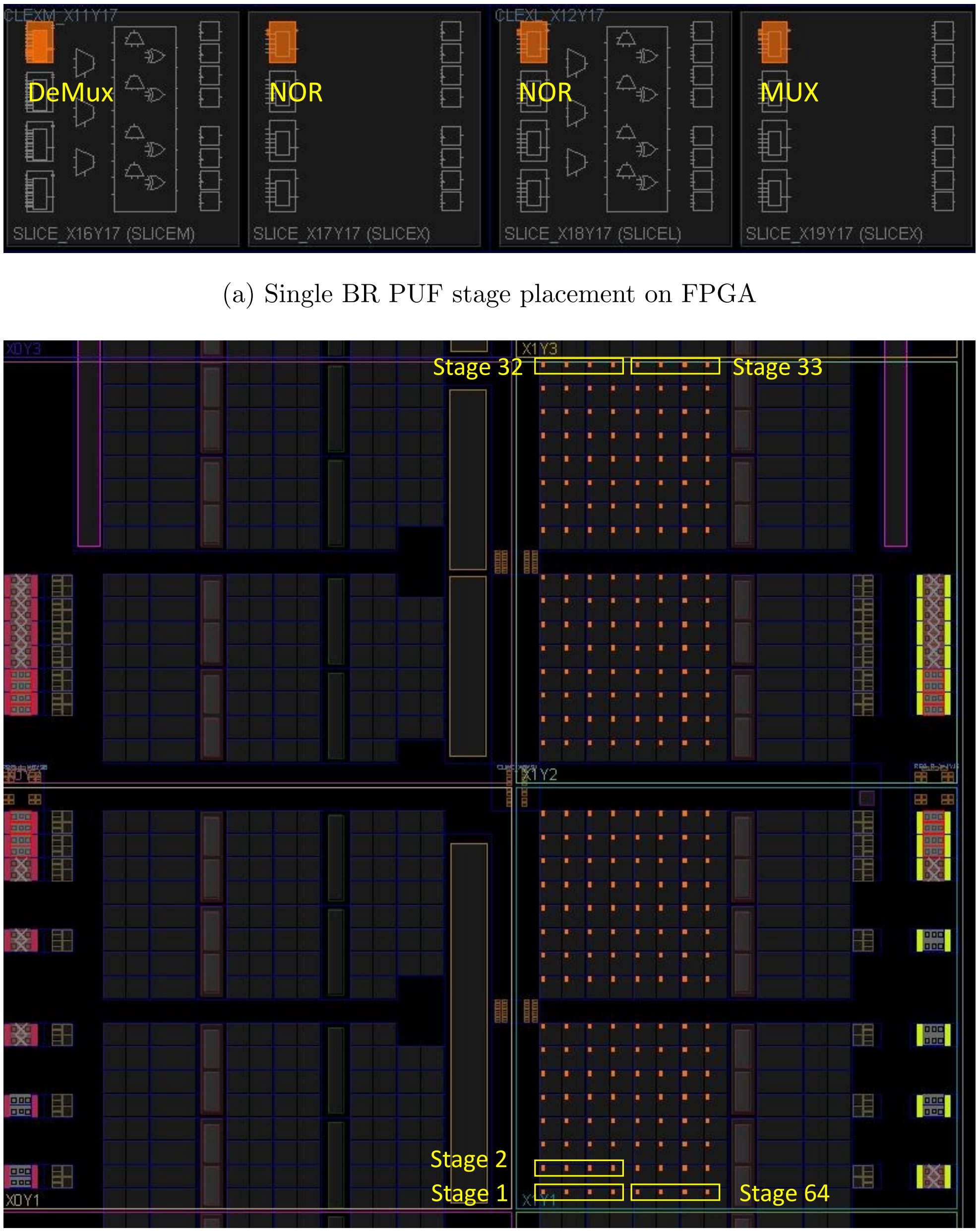
6.4.2 Manual Placement Approach

Various manual placement layouts were examined, all giving biased responses. Thus, only one layout will be discussed in this thesis, which is the same used in [76] on the same FPGA family. As Figure 6.6a shows, the two LUTs of the DeMux are combined in one LUT with two outputs, each NOR gate and the MUX are separated in different slices to ease up routing. The same stage layout is repeated 64 times as shown in Figure 6.6b.

The manual placement approach was also tested using the same five settling times producing the results in Table 6.4. Unlike with the automatic approach, the percentage of the stabilized responses increases with the increase of the settling time. The results also show a consistent bias that might slightly fluctuate due to the convergence of other responses with the increase of the settling time. Although one chip was very biased towards responses of 0, the other two had biases near 50%. However, the chips had a very low inter-chip hamming distance.

Table 6.4: Manual placement approach PUF characteristics

Metric	Settling Time (cycles)				
	64	128	256	512	1024
stabilized responses Chip 1 (%)	84	88	93	94	96
stabilized responses Chip 2 (%)	78	86	91	93	96
stabilized responses Chip 3 (%)	78	85	91	93	96
Bias Chip 1 (%)	20	18	18	24	25
Bias Chip 2 (%)	50	43	48	52	45
Bias Chip 3 (%)	48	38	45	53	42
Noise Chip 1 (%)	1.1	0.2	0.3	0.3	0.2
Noise Chip 2 (%)	1.5	0.5	0.3	0.2	0.3
Noise Chip 3 (%)	1.7	0.7	0.4	0.4	0.4
NHD Chips 1&2 (%)	43	34	36	32	23
NHD Chips 1&3 (%)	41	29	33	23	21
NHD Chips 2&3 (%)	18	18	16	14	13
NHD Average (%)	34	27	28	26	19
Max influence Chip 1 (%)	14	13	13	18	20
Max influence Chip 2 (%)	40	34	39	61	37
Max influence Chip 3 (%)	40	29	37	37	34



(a) Single BR PUF stage placement on FPGA

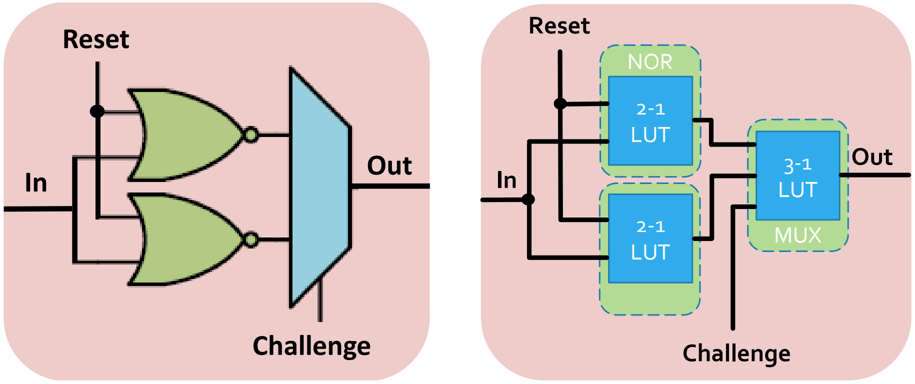
(b) 64 BR PUF stages placement on FPGA

Figure 6.6: Manual Placement Layout on FPGA for 64-bit BR PUF

6.5 Implementation of the 4-input XOR BR PUF

Although the automatic placement approach did not yield into an ideal PUF, it provided a way to manipulate the CAD tool to get a good one. And as for the XOR BR PUF there are many instances of the BR PUF XORed, which gives more room for the automatic placement to achieve more randomness and better properties. Something the manual placement approach will not be able to provide. Thus, the automatic placement approach was used for the XOR BR and TBR PUFs.

While the goal of this research is to implement a 4-input XOR BR PUF with 64, 128 and 256-bit challenges, only the 64-bit 4-input XOR BR PUF fits into the FPGA. Thus, an optimization was needed to minimize the number of LUTs of each BR PUF stage. The optimization was introduced to the architecture level by removing the DEMUX as in Figure 6.7a. As shown, eventually the MUX chooses between the output of either NOR gates, which is the important matter for the BR PUF functionality. Whether supplying the input to only one NOR gate as in the original design or to both NOR gates as in the optimized design, it would not make a difference to the BR PUF functionality. This optimization can reduce the number of LUTs in each stage from five to three as in Figure 6.7b.



(a) Optimized Single BR PUF Stage Schematic

(b) Optimized Single BR PUF Stage Layout

Figure 6.7: Optimized BR PUF Schematic and Layout on FPGA

Preliminary experiments in [33] [34] were conducted over the optimized and non-optimized implementations of the 64-bit BR PUF to empirically confirm that the optimization presented does not affect the security of the PUF. The obtained results showed that the optimized and non-optimized architectures have similar performances against both SVM and Deep learning modeling attack techniques.

Table 6.5: 4-input XOR BR PUF characteristics

Metric	PUF Size & Type 64-Bit Original	 64-Bit Optimized	 128-Bit Optimized	 256-Bit Optimized
Settling Time (Cycles)	1250	1024	4000	9600
stabilized responses Chip 1 (%)	83	81	81	86
stabilized responses Chip 2 (%)	81	82	83	85
stabilized responses Chip 3 (%)	80	79	82	85
Bias Chip 1 (%)	54	48	50	47
Bias Chip 2 (%)	52	49	49	54
Bias Chip 3 (%)	51	47	48	53
Noise Chip 1 (%)	1.2	1.7	2.0	1.2
Noise Chip 2 (%)	1.4	1.6	1.8	1.2
Noise Chip 3 (%)	1.4	1.7	1.9	1.2
NHD Chips 1&2 (%)	44	55	56	47
NHD Chips 1&3 (%)	50	48	51	50
NHD Chips 2&3 (%)	46	43	52	59
NHD Average (%)	47	53	52	50
Max influence Chip 1 (%)	58	46	51	45
Max influence Chip 2 (%)	54	45	51	58
Max influence Chip 3 (%)	53	43	47	56

The results in Table 6.5 show almost ideal PUFs with biases ranging from 47% to 54% around the ideal value of 50%. Noises are negligible with a maximum of 2%, noting that the unstabilized responses were excluded from any calculations. Also, no individual influential challenge bits exist, as the maximum influence of a challenge bit on any chip with all the PUF sizes is only 59%. The inter-chip hamming distance is around 50% which is the ideal value. The only drawback is the relatively low percentage of stabilized responses, which is around 80%. One of the reasons for this issue is that by XORing four different PUF instances if only one instance is not stabilized then the response of the XOR PUF will not be stabilized either, hence lower stabilized responses compared to a single PUF. The other reason is that BR PUFs have relatively low stabilized responses in general, which is a known drawback of the BR PUF family even for a single instance.

The implemented PUFs were tested against modeling attacks using both SVM with a polynomial kernel of degree four and deep learning in [33] [34]. Table 6.6 shows the modelling accuracy and the number of CRPs needed for both methods. The results shown for deep learning are for a fully connected neural network of 12 levels each comprising 2000 neurons, beside an input layer of M neurons where M is the challenge size of the PUF and an output layer of two neurons representing a one-hot decoding.

Table 6.6: 4-input XOR BR PUF modeling accuracy on Chip-1 using SVM and deep learning [33] [34]

Training Size	PUF Size, Type & Modeling Technique							
	64-Bit Original		64-Bit Optimized		128-Bit Optimized		256-Bit Optimized	
	DL	SVM	DL	SVM	DL	SVM	DL	SVM
500	56.3%	54.3%	50.1%	52.2%	50.1%	50.1%	53.2%	53.2%
1K	60.7%	55.4%	51.4%	52.7%	50.6%	50.3%	53.3%	53.3%
5K	67.8%	59.5%	67.6%	53.2%	51.8%	51.1%	88.8%	53.3%
10K	89.9%	61.7%	98.1%	54.1%	85.9%	51.8%	94.6%	53.3%
20K	95.5%	63.3%	99.2%	54.6%	96.7%	52.4%	96.4%	53.3%
50K	98.1%	64.5%	99.1%	55.2%	98.2%	53.7%	96.4%	53.4%
100K	98.6%	64.5%	98.1%	56.3%	98.8%	54.3%	96.7%	53.4%

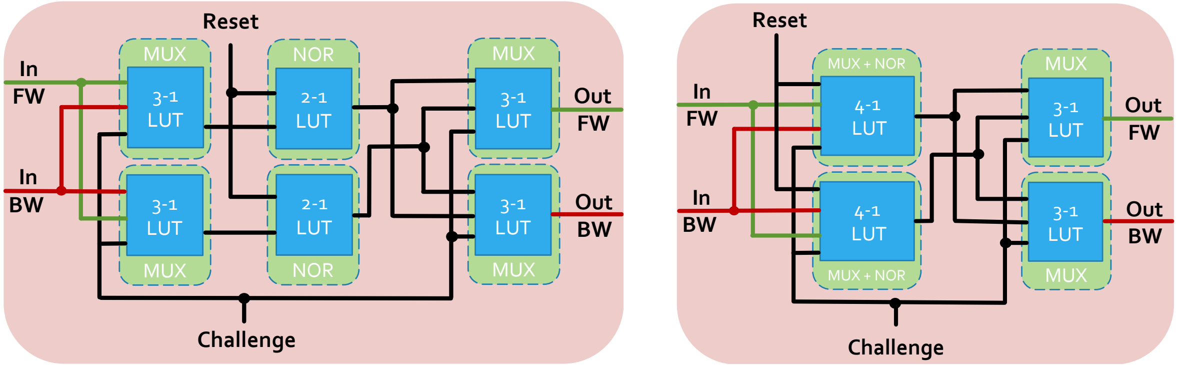
The results in [33] [34] show that the 4-input XOR BR PUFs of all sizes were resilient against SVM. However, deep learning was powerful enough to attack all the sizes with variant accuracy. Most importantly, the performances of the optimized and non-optimized implementations against the attacks are similar, evermore, the optimized implementation showed a slightly more resilience against SVM.

According to Table 6.6, an attacker needs as much as 10000 CRPs to guarantee a precise modelling (with more than 85% accuracy). While with even 1000 CRPs the deep learning performance would not differ much than SVM. Thus, to overcome such a powerful attack, 1000 CRPs could be set as an upper bound for the number of CRPs an adversary can collect before the PUF stops sending responses.

6.6 Implementation of the 4-input XOR TBR PUF

After the success of the automatic approach with the XOR BR PUF, the same approach was used with the TBR PUF. With the same problem of the 128 and 256-bit versions not fitting within the FPGA, an optimization on the TBR PUF implementation was needed.

Although there was no optimization possible on the architecture level, an optimization on the LUT layout level was achievable. The optimization represented in Figure 6.8b merges each NOR gate with its preceding MUX compared to the straightforward implementation in Figure 6.8a, resulting in a total of only four LUTs for one TBR PUF stage instead of six. This optimization does not affect the functionality of the TBR PUF as it maintains two inverting elements and two different paths, in addition to the fact that the merged LUT actually has the same logic of the two separate LUTs combined.



(a) Straightforward Single TBR PUF Stage Layout

(b) Optimized Single TBR PUF Stage Layout

Figure 6.8: Straightforward and Optimized TBR PUF Layouts on FPGA

As with the BR PUF, preliminary experiments were conducted in [33] [34] over the optimized and non-optimized implementations of the 64-bit TBR PUF showing similar performances against both SVM and Deep learning techniques. Thus, empirically confirming that the optimization presented does not affect the security of the PUF

The results in Table 6.7 show similar characteristics with the BR PUFs in terms of the almost ideal bias and inter-chip hamming distance, the lack of influential challenge bits and, the negligible noise. However, a slightly higher noise and a lower stabilized responses percentage for the TBR PUFs over the BR PUFs can be noticed. This could be explained by the fact that an N-bit TBR PUF holds a ring of 2N inverters compared to only N inverters in an N-bit BR PUF. This can also be noticed through the larger settling times for the TBR PUFs.

Table 6.7: 4-input XOR TBR PUF characteristics

Metric	PUF Size & Type 64-Bit Original	 64-Bit Optimized	 128-Bit Optimized	 256-Bit Optimized
Settling Time (Cycles)	6600	6000	13400	19000
stabilized responses Chip 1 (%)	81	72	78	68
stabilized responses Chip 2 (%)	87	76	67	67
stabilized responses Chip 3 (%)	77	73	74	73
Bias Chip 1 (%)	51	51	47	53
Bias Chip 2 (%)	51	54	52	47
Bias Chip 3 (%)	50	53	53	49
Noise Chip 1 (%)	2.0	3.2	2.6	3.1
Noise Chip 2 (%)	2.0	3.2	3.2	3.0
Noise Chip 3 (%)	2.4	3.4	3.0	2.7
NHD Chips 1&2 (%)	44	52	49	54
NHD Chips 1&3 (%)	44	55	46	55
NHD Chips 2&3 (%)	45	44	47	46
NHD Average (%)	44	50	47	52
Max influence Chip 1 (%)	57	55	44	53
Max influence Chip 2 (%)	57	59	56	45
Max influence Chip 3 (%)	55	59	55	46

The same modelling techniques were applied on the TBR PUF to examine its resilient against modelling attacks. The TBR PUF also was not attackable with the SVM but the deep learning technique was able to model it with all the PUF sizes. The only difference is in the number of CRPs needed to reach the same precision. As for the BR PUF, only 10000 CRPs were needed to guarantee 85% prediction accuracy. For the TBR PUF, 20000 CRPs were needed.

Table 6.8: 4-input XOR TBR PUF modeling accuracy on Chip-1 using SVM and deep learning [33] [34]

Training Size	PUF Size, Type & Modeling Technique							
	64-Bit		64-Bit		128-Bit		256-Bit	
	Original		Optimized		Optimized		Optimized	
	DL	SVM	DL	SVM	DL	SVM	DL	SVM
500	61.2%	53.6%	54.3%	52.5%	56.9%	52.4%	57.5%	54.2%
1K	66.5%	56.8%	54.5%	53.6%	58.8%	52.4%	57.7%	54.9%
5K	90.9%	62.2%	84.4%	56.7%	63.3%	53.7%	71.5%	59.4%
10K	95.3%	64.7%	85.9%	58.5%	76.4%	55.3%	87.2%	60.9%
20K	96.9%	66.9%	94.2%	59.8%	85.7%	57.7%	94.9%	62.4%
50K	98.2%	68.7%	97.1%	60.6%	95.6%	59.4%	97.1%	63.5%
100K	98.6%	68.7%	97.3%	60%	96.7%	62.5%	97.7%	62.9%

Chapter 7

Conclusion

This chapter concludes the thesis by giving a brief summary of this research in Section 7.1. It also discusses the research outcomes in Section 7.2, states the main research contributions in Section 7.3, and finally provides potential research directions for future work in Section 7.4.

7.1 Summary

In this thesis two different topics are presented under the embedded systems security umbrella. The first studies EM fault injection on a RISC-V processor, and the other investigates the implementation approaches for BR and TBR PUFs on FPGA.

For the EM fault injection part, an EMFI setup centered on Langer EMV-Technik's burst power station BPS-201 and its EM probe was used. The setup has between 284 ns and 296 ns delay from the trigger signal until the pulse is generated, with a rise time of 12 ns. The target board is a HiFive1 Rev B development board featuring a single-issue in-order execution five-stage pipeline RV32IMAC RISC-V core with a maximum functional frequency of 320 MHz.

The experiments started by scanning the chip with different EM pulse voltages to identify the best location over the chip where faults are more likely to be injected. Experiments were also conducted on the chip with various operating conditions in terms of clock frequency and supplied voltage, and it was confirmed that the higher the frequency and the lower the supplied voltage, the easier faults can be injected.

A timing analysis was performed to identify the delay needed between the trigger assertion in the code and the target instruction. It was found that the 110$^{\text{th}}$ instruction located after the trigger assertion is the one with the highest probability of having faults injected. It was also found that multiple instructions can be faulted at once. In fact, results showed that up to six instructions can be faulted at once.

Experiments were performed to analyze the effect of EMFI on different instructions, targeting arithmetic and logic operations, memory operations and flow control operations. Results show that arithmetic and logic operations respond to EMFI in a similar way to memory operations. On average around 50% of the time, the instructions are performed without faults, around 30% of the time instructions are skipped, and the remaining 20% results in corrupted data or the surrounding instructions get faulted causing exception faults. On the other hand, EMFI on unconditional jump instructions corrupt the return address linking with a probability of 28%. Interesting results for conditional branch instructions show that instructions do not get skipped, however, in scenarios where branching conditions are not met, 19% of the time the branch is taken while it is not expected to.

For the PUF part, hardware implementations of the BR and TBR PUFs were realized on Xilinx Spartan-6 FPGA. The manual and automatic placement approaches were examined on the BR PUF, leading to the conclusion that it is hard to implement a non-biased BR PUF on FPGAs due to the asymmetry in the connections between different LUTs on FPGA. On the other hand, although the automatic routing yielded to biased implementations, it was found that any small change in the RTL code such as changing the settling time results into a whole different implementation on the layout level, hence different PUF characteristics. Which can be used as a randomization factor to reach a non-biased PUF after multiple trials.

The automatic routing approach was effective in implementing 64, 128 and 256-bit 4-input XOR BR and TBR PUFs with almost ideal characteristics due to the increased randomness provided by having multiple PUF instances. Though, due to the limited resources on the used FPGA, optimizations were conducted on the architecture and layout levels to reduce the number of LUTs utilized specially for the 128 and 256-bit PUF implementations. The Architectural optimizations on the BR PUF reduced the number of utilized LUTs from five to three, while the layout optimizations on the TBR PUF reduced the number of utilized LUTs from six to four.

7.2 Discussion

The results presented in this research for the EMFI on RISC-V are interesting in many ways. Starting with the locations over the chip where faults are injected as shown in Figure 5.1 in Section 5.1, both locations are at the corners of the chip. Knowing that the chip is 6 X 6 mm and the die is 3 X 3 mm centered at the middle of the chip, it raises doubts that the EM pulses were not pointed to the die, but rather on the wiring.

- The bottom left corner hosts the VDD, IVDD, and the QSPI CLK pins. Since the QSPI is connected to the flash memory and the faults are injected to the codes run from the cache, the QSPI CLK pin or wiring is excluded. Also, the IVDD pin is supplying the IO complex, which should not affect the core and cause the reported fault. Thus, one of the explanations is that the EM pulse affected the VDD pin or its wiring to the die causing a voltage glitch, which is the cause of the faults injected.
- The top right corner hosts the RESET pin as well as the AON (Always ON) IVDD pin. Similarly, the EM pulse might have affected any of the two signals and caused a glitch.
- Another explanation is that maybe the probe is not perfectly perpendicular to the chip causing the magnetic flux to be aimed at the die while in the shmoo plots it seems not, or the edge of the probe (which might be on the die) is causing the fault injection.

The data visualized in both Figures 5.2 and 5.3 in Section 5.2 indicates that the processor is more susceptible to EMFI with the increase of clock frequency and with the decrease in supplied voltage, which complies with the charge-based fault model introduced in [40].

The BPS-201 burst power station showed a variant jitter and trigger-to-pulse delay depending on various conditions. For example, if the system is turned on for a long time, it gets heated and that affects its jitter and delay. Thus, the results in Section 5.3 and Section 5.4 might be slightly different than the ones reported in [21] performing the same experiments. Similarly, the experiments in Section 5.5 can yield different results than presented if conducted in a different time under different conditions. However, these variations are small but affect the accuracy of the fault injection.

The results reported in Section 5.4 are very interesting, showing that multiple consecutive instructions (up to six instructions) can be faulted at once. This can be very devastating to developers, as it makes software countermeasures like instruction duplication or triplication infeasible.

While the target processor is treated as a black box, with not much knowledge of the actual implementation in silicon, one cannot determine for sure the effect of the EM pulse on the core. However, possible explanations for having multiple instructions being skipped or faulted at once could be portrayed as follows:

- The long rise time of the generated EM pulse as displayed previously in Figure 4.4 in Section 4.1, which ranges from seven ns to 12 ns can roughly cover four instructions working with 320 MHz clock frequency. Thus, could be responsible for the multiple instructions skips.
- The fact that the target is a five-stage pipelined processor, and the ability of the EM pulse to have a global effect, makes it possible to affect up to five different instructions in the different five pipeline stages.
- By combining the two previous hypotheses, assuming that five instructions in the five pipeline stages are affected, in addition to the effect of the rise time that can cover four instructions, which effectively can add only three more instructions, then up to eight instructions can be hypothetically be faulted at once.

The experiments conducted in Section 5.5 showed different vulnerabilities against EMFI on various instructions. These vulnerabilities can be exploited to apply practical attacks on security primitives such as AES, RSA, ECC, or operating system's secure boot. The ability to skip arithmetic instructions such as ADD instruction can help attacking many ciphers like by manipulating the encryption round numbers causing the skip of rounds, or exploiting the XOR and instruction to corrupt the key XORing with the state in AES causing a side-channel leakage. Also, the ability to fault branches can be used to attack secure boot and skip authentication routines in general.

For the BR PUF design, the investigation of the manual placement and automatic placement implementation approaches resulted in interesting conclusions. The manual placement approach showed that due to the layout constraints on the FPGA and the prefabricated interconnections of the FPGA, it is very hard to design a non-biased BR PUF on FPGA since the BR PUF requires symmetry to ensure a non-biased design and there will always be variations by the design in the different ring paths.

While the automatic placement approach on a single BR PUF produced biased PUFs as with the manual placement approach, however, it was noticed that with the change of the settling time of the PUF the whole placement procedures are run again differently, producing different PUFs with different characteristics. Through multiple trials with different settling times, a non-biased PUF can be produced, which was efficient in the design of the 4-input BR and TBR PUFs. Not only settling time, any other modification on the HDL code would force the re-run of the placement procedures. Even a trivial block could be implemented with the PUF on FPGA, and that block can be changed to force the re-run of the placement without changing the PUF settling time.

The BR PUF design introduced in [13] comprised of a DEMUX, 2 NOR gates and a MUX for each stage as shown in Figure 6.4 in Section 6.3. However, this DEMUX seems to be redundant, as the MUX eventually chooses between either NOR gates' output. Hence, the optimization in Figure 6.7 in Section 6.5 be removing the DEMUX was perfromed with the expectation of not effecting the PUF's security, which was confirmed by the modeling attacks performed in [34].

As the results of the modeling attacks in Table 6.6 and Table 6.8 show that an attacker needs as much as 10000 CRPs to guarantee a precise modelling. Thus, a simple countermeasure to overcome such a powerful attack, is to set 10000 CRPs as an upper bound for the number of CRPs an entity can collect before the PUF stops sending responses.

7.3 Contributions

While EMFI on processors might have been discussed in previous research as well as the implementation of BR PUF on FPGA, the novelty and the main contributions of this thesis could be listed as follows:

1. This research is the first to apply EMFI on a RISC-V processor [21]. In this research, an analysis of the EM effect on different instructions has been conducted. As well as an exploration of the different conditions which makes the processor more susceptible

to fault injection. The procedures used and outcomes can help in studying EMFI on other processors as well.

2. As per our knowledge, this research is the first to report multiple instructions being faulted at once using EMFI. which can be very devastating as discussed earlier.

3. Most of the previous research regarding BR and TBR PUFs did not detail their implementations on FPGA or the approaches used. In this thesis, the FPGA implementations are detailed and different approaches are discussed and compared.

4. The introduction of the settling time or any minor change in the RTL code as a randomization factor to achieve good PUF characteristics is one of the contributions of this research, enabling a practical approach to implement PUFs on FPGAs.

5. Both BR and TBR PUFs are optimized on architectural and layout level in this research. Which reduces the size of the implemented PUFs and enables multiple PUF instances to be implemented and XORed to increase the PUF security.

7.4 Future Work

As this research focused more on the analysis of the EMFI effect on RISC-V, no real attacks were performed in this thesis. However, future work can include the exploitation of the vulnerabilities analyzed. These exploitations can work on compromising ciphers or authentication routines such as in operating systems' secure boot.

Also, since faulting multiple instructions at once can compromise the software countermeasures, this drives a research direction towards applying hardware countermeasures, either on the instruction level or the architecture level. As RISC-V is open source and the RTL code of the core used in this research is available online, an emulation of the same core on FPGA is achievable, as well as the possibility to develop and test various hardware countermeasures on the RISC-V architecture on FPGA.

For the BR/TBR PUF design on FPGA, various countermeasures can be applied and tested, such as challenge bits obfuscation to complicate the relationship between the challenges and the responses to an attacker as discussed in [34].

Another research direction would be to apply the same implementation approaches on different FPGA families such as Intel and Microsemi, and even different Xilinx FPGAs. This would help in understanding the effects of the different approaches for FPGA based PUFs in general without the risk of being specific to a certain FPGA family.

References

[1] Maurice Aarts. *Electromagnetic Fault Injection using Transient Pulse Injections*. PhD thesis, Master's thesis, Eindhoven University of Technology, Netherlands, 2013.

[2] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. When clocks fail: On critical paths and clock faults. In *International Conference on Smart Card Research and Advanced Applications*, pages 182–193. Springer, 2010.

[3] Peter Alfke. Efficient shift registers, lfsr counters, and long pseudo-random sequence generators. *http://www. xilinx. com/bvdocs/appnotes/xapp052. pdf*, 1998.

[4] Nikolaos Athanasios Anagnostopoulos. Optical fault injection attacks in smart card chips and an evaluation of countermeasures against them. Master's thesis, University of Twente, 2014.

[5] Siva Prashanth BALAMURALI. An improved public unclonable function design for xilinx fpgas for hardware security applications. 2018.

[6] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 105–114. IEEE, 2011.

[7] 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.

[8] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. Countermeasures against fault attacks on software implemented aes: effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security*, page 7. ACM, 2010.

[9] Todd Bauer and Jason Hamlet. Physical unclonable functions: A primer. *IEEE Security & Privacy*, 12(6):97–101, 2014.

[10] BK precision. *https://www.bkprecision.com*.

[11] Carbide 3D. *https://carbide3d.com*.

[12] Urbi Chatterjee, Rajat Subhra Chakraborty, and Debdeep Mukhopadhyay. A puf-based secure communication protocol for iot. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(3):67, 2017.

[13] Qingqing Chen, György Csaba, Paolo Lugli, Ulf Schlichtmann, and Ulrich Rührmair. The bistable ring puf: A new architecture for strong physical unclonable functions. In *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 134–141. IEEE, 2011.

[14] Qingqing Chen, György Csaba, Paolo Lugli, Ulf Schlichtmann, and Ulrich Rührmair. The bistable ring puf: A new architecture for strong physical unclonable functions. In *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 134–141. IEEE, 2011.

[15] Qingqing Chen, György Csaba, Paolo Lugli, Ulf Schlichtmann, and Ulrich Rührmair. Characterization of the bistable ring puf. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1459–1462. EDA Consortium, 2012.

[16] Ang Cui and Rick Housley. {BADFET}: Defeating modern secure boot using second-order pulsed electromagnetic fault injection. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.

[17] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic transient faults injection on a hardware and a software implementations of aes. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 7–15. IEEE, 2012.

[18] Amine Dehbaoui, Amir-Pasha Mirbaha, Nicolas Moro, Jean-Max Dutertre, and Assia Tria. Electromagnetic glitch on the aes round counter. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 17–31. Springer, 2013.

[19] Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. Technical report, 2015.

[20] Schuyler Eldridge, Alper Buyuktosunoglu, and Pradip Bose. Chi re: A configurable hardware fault injection framework for risc-v systems. 2018.

[21] Mahmoud A. Elmohr, Haohao Liao, and Catherine H. Gebotys. Em fault injection on arm and risc-v. To appear in *21st International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2020.

[22] eMemory. *https://www.ememory.com.tw*.

[23] Thomas Finke, Max Gebhardt, and Werner Schindler. A new side-channel attack on rsa prime generation. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 141–155. Springer, 2009.

[24] Fatemeh Ganji, Shahin Tajik, Fabian Faessler, and Jean-Pierre Seifert. Having no mathematical model may not secure pufs. *Journal of Cryptographic Engineering*, 7(2):113–128, 2017.

[25] Fatemeh Ganji, Shahin Tajik, Fabian Fäßler, and Jean-Pierre Seifert. Strong machine learning attack against pufs with no mathematical model. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 391–411. Springer, 2016.

[26] Fatemeh Ganji, Shahin Tajik, and Jean-Pierre Seifert. Why attackers win: on the learnability of xor arbiter pufs. In *International Conference on Trust and Trustworthy Computing*, pages 22–39. Springer, 2015.

[27] Blaise Gassend, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 148–160. ACM, 2002.

[28] Blaise Laurent Patrick Gassend. *Physical random functions*. PhD thesis, Massachusetts Institute of Technology, 2003.

[29] Marjan Ghodrati. *Thwarting electromagnetic fault injection attack utilizing timing attack countermeasure*. PhD thesis, Virginia Tech, 2018.

[30] Jorge Guajardo, Sandeep S Kumar, Geert-Jan Schrijen, and Pim Tuyls. Fpga intrinsic pufs and their use for ip protection. In *International workshop on cryptographic hardware and embedded systems*, pages 63–80. Springer, 2007.

[31] Intel. *https://www.intel.com*.

[32] Intrinsic ID. *https://www.intrinsic-id.com*.

[33] Mahmoud KhalafAlla, Mahmoud A. Elmohr, and Gebotys. Going deep: Using deep learning techniques with simplified mathematical models against xor br and tbr pufs (attacks and countermeasures). To appear in *2020 IEEE International Symposium on Hardware-Oriented Security and Trust*. IEEE, 2020.

[34] Mahmoud Khalafallah. Comprehensive study of physical unclonable functions on fpgas: correlation driven implementation, deep learning modeling attacks, and countermeasures. 2020.

[35] Paul Kocher, Ruby Lee, Gary McGraw, Anand Raghunathan, Srivaths Moderator-Ravi, and Srivaths Moderator-Ravi. Security as a new dimension in embedded system design. In *Proceedings of the 41st annual Design Automation Conference*, pages 753–760. ACM, 2004.

[36] Oliver Kömmerling and Markus G Kuhn. Design principles for tamper-resistant smartcard processors. *Smartcard*, 99:9–20, 1999.

[37] Philip Koopman. Embedded system security. *Computer*, 37(7):95–97, 2004.

[38] LANGER EMV-Technik GmbH. *https://www.langer-emv.com.*

[39] 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.

[40] Haohao Liao and Catherine Gebotys. Methodology for em fault injection: Charge-based fault model. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 256–259. IEEE, 2019.

[41] Daihyun Lim, Jae W Lee, Blaise Gassend, G Edward Suh, Marten Van Dijk, and Srinivas Devadas. Extracting secret keys from integrated circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(10):1200–1205, 2005.

[42] Roel Maes and Ingrid Verbauwhede. Physically unclonable functions: A study on the state of the art and future research directions. In *Towards Hardware-Intrinsic Security*, pages 3–37. Springer, 2010.

[43] Microsemi. *https://www.microsemi.com.*

[44] Victor S Miller. Use of elliptic curves in cryptography. In *Conference on the theory and application of cryptographic techniques*, pages 417–426. Springer, 1985.

[45] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 77–88. IEEE, 2013.

[46] Nicolas Moro, Karine Heydemann, Amine Dehbaoui, Bruno Robisson, and Emmanuelle Encrenaz. Experimental evaluation of two software countermeasures against fault attacks. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 112–117. IEEE, 2014.

[47] Erick Nascimento, Łukasz Chmielewski, David Oswald, and Peter Schwabe. Attacking embedded ecc implementations through cmov side channels. In *International Conference on Selected Areas in Cryptography*, pages 99–119. Springer, 2016.

[48] NXP. *https://www.nxp.com.*

[49] Sébastien Ordas, Ludovic Guillaume-Sage, and Philippe Maurine. Electromagnetic fault injection: the curse of flip-flops. *Journal of Cryptographic Engineering*, 7(3):183–197, 2017.

[50] Srivaths Ravi, Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. Security in embedded systems: Design challenges. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(3):461–491, 2004.

[51] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[52] Lionel Riviere, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. High precision fault injections on the instruction cache of armv7-m architectures. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 62–67. IEEE, 2015.

[53] MAES Roel. Physically unclonable functions: Constructions, properties and applications. *Katholieke Universiteit Leuven, Belgium*, 2012.

[54] Rohde & Schwarz. *https://www.rohde-schwarz.com.*

[55] Ulrich Rührmair and Daniel E Holcomb. Pufs at a glance. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 347. European Design and Automation Association, 2014.

[56] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. Modeling attacks on physical unclonable functions. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 237–249. ACM, 2010.

[57] Dieter Schuster and Robert Hesselbarth. Evaluation of bistable ring pufs using single layer neural networks. In *International Conference on Trust and Trustworthy Computing*, pages 101–109. Springer, 2014.

[58] Dieter Schuster and Robert Hesselbarth. Evaluation of bistable ring pufs using single layer neural networks. In *International Conference on Trust and Trustworthy Computing*, pages 101–109. Springer, 2014.

[59] SEGGER. *https://www.segger.com*.

[60] Nidhal Selmane, Sylvain Guilley, and Jean-Luc Danger. Practical setup time violation attacks on aes. In *2008 Seventh European Dependable Computing Conference*, pages 91–96. IEEE, 2008.

[61] SiFive. *https://www.sifive.com*.

[62] SiFive, Inc. *SiFive E31 Manual*, 9 2019. v19.08p0.

[63] SiFive, Inc. *SiFive FE310-G002 Manual*, 5 2019. v19p05.

[64] SiFive, Inc. *SiFive HiFive1 Rev B Getting Started Guide*, 5 2019. V1.1.

[65] Sergei Skorobogatov. Using optical emission analysis for estimating contribution to power analysis. In *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 111–119. IEEE, 2009.

[66] Sergei Petrovich Skorobogatov. Semi-invasive attacks: a new approach to hardware security analysis. 2005.

[67] Sudheendra Srivathsa. Secure and energy efficient physical unclonable functions. 2012.

[68] Advance Encryption Standard. Federal information processing standards publication 197. *FIPS PUB*, pages 46–3, 2001.

[69] G Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *2007 44th ACM/IEEE Design Automation Conference*, pages 9–14. IEEE, 2007.

[70] 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.

[71] Roy Ward and Tim Molteno. Table of linear feedback shift registers. *Datasheet, Department of Physics, University of Otago*, 2007.

[72] 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.

[73] xilinx. *https://www.xilinx.com.*

[74] Xiaolin Xu, Ulrich Rührmair, Daniel E Holcomb, and Wayne Burleson. Security evaluation and enhancement of bistable ring pufs. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 3–16. Springer, 2015.

[75] Dai Yamamoto, Masahiko Takenaka, Kazuo Sakiyama, and Naoya Torii. Security evaluation of bistable ring pufs on fpgas using differential and linear analysis. In *2014 Federated Conference on Computer Science and Information Systems*, pages 911–918. IEEE, 2014.

[76] Dai Yamamoto, Masahiko Takenaka, Kazuo Sakiyama, and Naoya Torii. Security evaluation of bistable ring pufs on fpgas using differential and linear analysis. In *2014 Federated Conference on Computer Science and Information Systems*, pages 911–918. IEEE, 2014.

[77] Tanveer Zia and Albert Zomaya. Security issues in wireless sensor networks. In *2006 International Conference on Systems and Networks Communications (ICSNC'06)*, pages 40–40. IEEE, 2006.

[78] Loic Zussa, Amine Dehbaoui, Karim Tobich, Jean-Max Dutertre, Philippe Maurine, Ludovic Guillaume-Sage, Jessy Clediere, and Assia Tria. Efficiency of a glitch detector against electromagnetic fault injection. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014.

[79] Loic Zussa, Jean-Max Dutertre, Jessy Clédiere, Bruno Robisson, Assia Tria, et al. Investigation of timing constraints violation as a fault injection means. In *27th Conference on Design of Circuits and Integrated Systems (DCIS), Avignon, France*. Citeseer, 2012.

APPENDICES

Appendix A

BR/TBR PUFs Codes

A.1 MUX Module

Code A.1: MUX Module RTL Code in Verilog

```verilog
module Mux(
    input in_0,       //first input to the MUX
    input in_1,       //second input to the MUX
    input sel,        //the MUX's selector
    output out        //the MUX's output
);

    assign out = sel==1? in_1 : in_0;   //definition of the MUX

endmodule
```

A.2 DE-MUX Module

Code A.2: DE-MUX Module RTL Code in Verilog

```
module DeMux(
    input in,       //the DEMUX’s intput
    input sel,      //the DEMUX’s selector
    output out_0,   //first output of the DEMUX
    output out_1    //second output of the DEMUX
);

    assign out_0 = sel==1? 1’b0 : in;    //definition of the DEMUX
    assign out_1 = sel==1? in : 1’b0;    //definition of the DEMUX

endmodule
```

A.3 Original Single BR PUF Stage Module

Code A.3: Original Single BR PUF Stage Module RTL Code in Verilog

```verilog
module BRStage(
    input rst,          //reset signal
    input challenge,    //corresponding challenge bit
    input in,           //input signal to the stage
    output out          //output signal of the stage
);

    wire nor_0_in;  //1st output of the DEMUX and input to 1st nor gate
    wire nor_1_in;  //2nd output of the DEMUX and input to 2nd nor gate
    wire nor_0_out; //output of 1st nor gate and 1st input to the MUX
    wire nor_1_out; //output of 2nd nor gate and 2nd input to the MUX

    //makes sure the DEMUX is not removed for optimizations
    (* KEEP_HIERARCHY = "true" *)
    DeMux DeMux_inst (  //instantiates the DEMUX and connects its port
        .in(in),
        .sel(challenge),
        .out_0(nor_0_in),
        .out_1(nor_1_in)
    );

    //instantiate the nor gate and connect their port
    nor nor_0(nor_0_out,nor_0_in,rst);
    nor nor_1(nor_1_out,nor_1_in,rst);

    //makes sure the MUX is not removed for optimizations
    (* KEEP_HIERARCHY = "true" *)
    Mux Mux_inst (  //instantiates the MUX and connects its port
        .in_0(nor_0_out),
        .in_1(nor_1_out),
        .sel(challenge),
        .out(out)
    );

endmodule
```

A.4 Optimized Single BR PUF Stage Module

Code A.4: Optimized Single BR PUF Stage Module RTL Code in Verilog

```
module BRStage(
    input rst,          //reset signal
    input challenge,    //corresponding challenge bit
    input in,           //input signal to the stage
    output out          //output signal of the stage
);

    //makes sure both nor gates are not removed for optimizations
    (* KEEP = "true" *) wire nor_0_out; //output of 1st nor gate
    (* KEEP = "true" *) wire nor_1_out; //output of 2nd nor gate

    //instantiate the nor gate and connect their port
    nor nor_0(nor_0_out,in,rst);
    nor nor_1(nor_1_out,in,rst);

    //no need to explicitly keep the hierarchy of the MUX since both nor
   gates are kept, so the MUx won't be removed
    Mux Mux_inst (  //instantiates the MUX and connects its port
        .in_0(nor_0_out),
        .in_1(nor_1_out),
        .sel(challenge),
        .out(out)
    );

endmodule
```

A.5 BR PUF Top Module

Code A.5: BR PUF Top Module RTL Code in Verilog

```
module PUFTop#(parameter CHALLENGE_WIDTH = 64)(
    input rst,                              //reset signal
    input [CHALLENGE_WIDTH-1:0] challenge,  //full challenge
    output [CHALLENGE_WIDTH-1:0] response   //response of all stages
);

    generate

        wire [CHALLENGE_WIDTH-1:0] in_out;  //signals between stages
        genvar i;

        for (i = 0; i < CHALLENGE_WIDTH; i=i+1) begin : stage
            BRStage stage_inst (    //instantiates one BR stage
                .rst(rst),
                .challenge(challenge[i]),
                .in(in_out[i]),
                //connects output of one stage to the input of next
                .out(in_out[(i+1)%CHALLENGE_WIDTH])
            );
        end

        //assigns the final response with the signals between stages
        assign response = in_out;

    endgenerate

endmodule
```

A.6 Original Single TBR PUF Stage Module

Code A.6: Original Single TBR PUF Stage Module RTL Code in Verilog

```verilog
module TBRStage(
    input rst,          //reset signal
    input challenge,    //corresponding challenge bit
    input in_0,         //forward path input to the stage
    input in_1,         //backward path input to the stage
    output out_0,       //forward path output of the stage
    output out_1        //backward path output of the stage
);

    //makes sure all muxes and gates are kept separate and not optimized
    (* KEEP = "true" *) wire nor_0_in;  //input to the 1st nor gate
    (* KEEP = "true" *) wire nor_1_in;  //input to the 2nd nor gate
    (* KEEP = "true" *) wire nor_0_out; //output to the 1st nor gate
    (* KEEP = "true" *) wire nor_1_out; //output to the 2nd nor gate

    //instantiate the nor gate and connect their port
    nor nor_0(nor_0_out,nor_0_in,rst);
    nor nor_1(nor_1_out,nor_1_in,rst);

    Mux InMux_0 (   //instantiates the 1st input MUX
        .in_0(in_0),        .in_1(in_1),
        .sel(challenge),    .out(nor_0_in)
    );

    Mux InMux_1 (   //instantiates the 2nd input MUX
        .in_0(in_1),        .in_1(in_0),
        .sel(challenge),    .out(nor_1_in)
    );

    Mux OutMux_0 (  //instantiates the 1st output MUX
        .in_0(nor_0_out),   .in_1(nor_1_out),
        .sel(challenge),    .out(out_0)
    );

    Mux OutMux_1 (  //instantiates the 2nd output MUX
        .in_0(nor_1_out),   .in_1(nor_0_out),
        .sel(challenge),    .out(out_1)
    );

endmodule
```

A.7 Optimized Single TBR PUF Stage Module

Code A.7: Optimized Single TBR PUF Stage Module RTL Code in Verilog

```
module TBRStage(
    input rst,          //reset signal
    input challenge,    //corresponding challenge bit
    input in_0,         //forward path input to the stage
    input in_1,         //backward path input to the stage
    output out_0,       //forward path output of the stage
    output out_1        //backward path output of the stage
);

    //the inputs of the nor gates are left for optimization
    wire nor_0_in;                      //input to the 1st nor gate
    wire nor_1_in;                      //input to the 2nd nor gate
    (* KEEP = "true" *) wire nor_0_out; //output to the 1st nor gate
    (* KEEP = "true" *) wire nor_1_out; //output to the 2nd nor gate

    //instantiate the nor gate and connect their port
    nor nor_0(nor_0_out,nor_0_in,rst);
    nor nor_1(nor_1_out,nor_1_in,rst);

    Mux InMux_0 (   //instantiates the 1st input MUX
        .in_0(in_0),        .in_1(in_1),
        .sel(challenge),    .out(nor_0_in)
    );

    Mux InMux_1 (   //instantiates the 2nd input MUX
        .in_0(in_1),        .in_1(in_0),
        .sel(challenge),    .out(nor_1_in)
    );

    Mux OutMux_0 (  //instantiates the 1st output MUX
        .in_0(nor_0_out),   .in_1(nor_1_out),
        .sel(challenge),    .out(out_0)
    );

    Mux OutMux_1 (  //instantiates the 2nd output MUX
        .in_0(nor_1_out),   .in_1(nor_0_out),
        .sel(challenge),    .out(out_1)
    );

endmodule
```

A.8 TBR PUF Top Module

Code A.8: TBR PUF Top Module RTL Code in Verilog

```
module PUF_chain#(parameter CHALLENGE_WIDTH = 64)(
    input rst,                               //reset signal
    input [CHALLENGE_WIDTH-1:0] challenge,  //full challenge
    output [2*CHALLENGE_WIDTH-1:0] response //response of all stages
);

    generate
        wire [2*CHALLENGE_WIDTH-1:0] in_out;    //signals between stages
        genvar i;
        for (i = 0; i < CHALLENGE_WIDTH; i=i+1) begin : stage
            TBRStage stage_inst (    //instantiates one TBR stage
            .rst(rst),
            .challenge(challenge[i]),
            .in_0(in_out[i]),
            .in_1(in_out[2*CHALLENGE_WIDTH-i-1]),
            .out_0(in_out[i+1]),
            .out_1(in_out[(2*CHALLENGE_WIDTH-i)%(2*CHALLENGE_WIDTH)])
            );
        end

        //assigns the final response with the signals between stages
        assign response = in_out;

    endgenerate

endmodule
```