Extending FiSim, a Fault Attack Simulator

Report by:
Mashal ZAINAB

Supervised by:
Ileana Buhan

In Partial Fulfillment of the Requirements for the
Degree of
Masters in Cybersecurity

UNIVERSITÉ BRETAGNE SUD
Lorient, France

Defended 27th June, 2023

Abstract

Fault injection attacks, which use the manipulation of operating conditions to impair a processor's normal operation, have grown increasingly sophisticated over time. The demand for reliable fault injection simulators has increased due to the prevalence of fault injection attacks and the development of cutting-edge techniques. These simulators allow researchers and practitioners to study and improve the security of their systems by simulating the complex environments needed to carry out fault injection attacks.

We explore the field of fault injection simulators in this extensive report, concentrating particularly on one tool called FiSim. FiSim was created to demonstrate how fault injection attacks work to get around secure boot mechanisms. It offers a flexible environment for researchers to investigate various fault injection methods and the effects they have on how programs run.

Our investigation includes a thorough analysis of the FiSim fault injection techniques. We delve into the fundamentals of fault injection and go over how FiSim makes use of these methods to inject controlled faults into specific programs. We carefully examine the fault models used by FiSim to simulate actual fault injection scenarios.

Despite FiSim's usefulness in fault injection research, we also discuss some of its limitations. We propose and create an extended version of the simulator, which we would refer to as FiSim++, building on the framework established by FiSim. The injection of faults into various cryptographic algorithms that are known to be susceptible to Differential Fault Analysis (DFA) attacks is one of the features we add to FiSim's capabilities in this improved iteration. Researchers can assess the resistance of cryptographic algorithms to DFA attacks using FiSim++ and spot potential flaws that need more investigation.

We assess three well-known cryptographic algorithms' resistance to fault injection attacks by subjecting FiSim++ to extensive testing. The outcomes of these tests shed important light on the performance of FiSim++ and the security of the examined cryptographic algorithms in fault injection scenarios.

We anticipate many scenarios for FiSim++ in the future that could increase its usefulness in the area of fault injection. FiSim++ can become an essential tool for researchers and practitioners by seizing these chances, enabling them to defend their systems from fault injection attacks and increase overall security.

Keywords:
FI (Fault Injection),
FiSim (Fault Injection Simulator),
FiSim++ (Extended version of Fault Injection Simulator),
AES (Advanced Encryption Standard),
DES (Data Encryption Standard),
SHA (Secure Hash Algorithm),
DFA (Differential Fault Attack),
ARM (Advanced RISC Machine),
ISA (Instruction Set Architecture)

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

LISTINGS

Chapter 1

INTRODUCTION

In an increasingly interconnected world, where information and communication technologies play a very pivotal role, ensuring the security and integrity of digital systems has become a paramount concern. As organizations and individuals become increasingly reliant on these systems for critical operations and sensitive data handling, the need to protect them from malicious activities becomes even more crucial. As technology evolves and defenses improve, so do the techniques employed by malicious actors. Among the most sophisticated and insidious methods utilized by adversaries, fault injection attacks have emerged as a significant concern.

Fault injection attacks represent a class of security vulnerabilities that exploit weaknesses in the design or implementation of a system, allowing attackers to compromise its integrity, availability, or security. These attacks involve deliberately introducing faults or errors into a target system, thereby disrupting its normal operation and potentially gaining unauthorized access or control. By manipulating hardware or software components, attackers can exploit vulnerabilities and bypass security mechanisms, leading to severe consequences. Fault injection attacks can be employed to bypass the secure boot process of a device, extract cryptographic keys, attack neural networks implementation etc.

The spectrum of fault injection attacks is vast, encompassing various techniques and vectors. For instance, attackers may induce faults by manipulating voltage levels, altering clock frequencies, introducing electromagnetic interference, make use of heating or power supplies or even employ software-based approaches. These methods are usually called glitches and are used to break the security of hardware or software systems. Each of these methods aims to disrupt the expected behavior of a system, opening avenues for exploitation and compromising its security objectives. In some cases, these glitches can also result in the damage of the hardware chips and devices as well.

The repercussions of successful fault injection attacks can be far-reaching. Fault injection attacks can be leveraged to subvert security mechanisms, tamper with critical processes, extract sensitive information, or gain unauthorized privileges. In technical terms, a fault or a glitch can modify memory contents, register contents,

change the executed instruction and as a result change the intended behaviour of software systems. Hence, it is significant and also mandated by certifications such as Common Criteria (CC) [8] etc. to test such systems against fault injection attacks in order to ensure the security across the system. And make sure that these systems cannot be attacked through fault injection attacks.

1.1 Fault Simulators

However, testing fault injection attacks in practice present a multitude of challenges that make it a complex task. These challenges include factors such as complexity and cost, safety concerns, lack of control and reproducibility, limited test coverage, and time inefficiency. One of the primary challenges in testing fault injection attacks is the complexity and cost associated with conducting real-world experiments. Physical access to the target system and specialized equipment may be required to manipulate hardware components or inject faults which can result in being very expensive. Another significant challenge is the lack of control and reproducibility in real-world testing scenarios. Precisely controlling the timing, location, and severity of injected faults can be exceedingly difficult, leading to uncertainties in evaluating the system's response and resilience. Additionally, conducting real-world fault injection tests can be time-consuming and inefficient. However, these challenges can be effectively mitigated by utilizing fault injection simulators, which serve as powerful tools to overcome the difficulties encountered during testing.

To address these challenges, fault injection simulators emerge as valuable tools for testing and evaluating the effectiveness of fault injection attacks. These simulators provide a controlled environment where testers can manipulate and inject faults with precision. A simulated environment to inject faults reduces the need to have complicated physical setups or equipment to perform fault injection attacks and enhance reproducibility by allowing testers to recreate the same fault injection conditions.

1.2 Related Work

There exist many fault simulation tools, possessing different functionalities and different purposes. And for the sake of this research, several Fault Injection tools were explored, varying from commercial, research and prototypes, in order to understand and gauge the vast range of Fault Injection techniques and approaches used around the world [1].

AWS Fault Injection Simulator [17] provides a managed service for running fault

injection experiments and enables controlled fault injection testing in AWS environments, allowing users to simulate various failure scenarios and evaluate the resiliency, performance and observability of their applications and systems on AWS infrastructure. By injecting faults like network latency and resource exhaustion, users can assess application behavior, identify vulnerabilities, and validate fault tolerance mechanisms.

Another well known tool called “Chaos Monkey” by Netflix [5] implements the concept of “Chaos Engineering”, to observe server failures and system crashes. Chaos engineering approach involves conducting experiments on a system to develop trust in its ability to handle challenging conditions during actual operation. The technique treats the distributed system as a collection of active functional units and focuses on monitoring and testing its dynamic behavior, known as "steady-state characteristics," by introducing realistic faults and errors.

Among other famous research tools, MODIFI (MODel-Implemented Fault Injection) [27] is specifically developed for the evaluation of robustness in Simulink behavior models which offers the capability to incorporate domain-specific fault models through XML, providing researchers and practitioners with a valuable resource for assessing the resilience and fault tolerance of Simulink-based systems.

Ferrari (Fault and ERRor Automatic Real-time Injection) [16], introduces the utilization of software traps to inject faults into a system. The faults can be of two types: transient, occurring temporarily, or permanent, persisting in the system. Extensive research conducted with Ferrari demonstrates that the effectiveness of error detection depends on the type of fault and its specific insertion location within the system and provides valuable insights into understanding error behavior and detection in various scenarios.

In addition to these famous tools, others that are closer to our research area include, ZOFI (Zero Overhead Fault Injection) [22], an innovative fault injection tool focused on studying transient faults and their fault coverage. And is specifically designed to operate with zero overhead, allowing the analyzed workload to run at its natural speed, offering a significant advantage over traditional methods.

ARMORY [14], is a fully automated and comprehensive open-source framework that enables exhaustive fault simulation on ARM-M class binaries. It provides engineers and analysts with an efficient way to scan binaries and identify vulnerabilities by subjecting them to various fault models and combinations of multivariate fault injections.

Introducing the concept of “user defined fault campaigns” and ARCHitecture-

Independent Evaluation of faults, ARCHIE [13] operates on QEMU, and facilitates the assessment of faults by allowing users to define fault campaigns through a JSON configuration file. Once configured, ARCHIE automatically executes the entire campaign without requiring further user input. Also providing support for the simulation of both permanent and transient faults in instructions, memory, and registers.

1.3 Motivation

We see that there are various high level and low level simulators, however, there is a need of an open source and simplified fault simulator which is able to simulate various programs for various platforms, specifically cryptographic algorithms and post quantum cryptographic algorithm. This is the motivation that we take along during this research and extend an existing training tool to support the above requirements, whose details will be explained in the upcoming chapters.

Chapter 2

FISIM

Secure Boot is a crucial component of the security framework in modern embedded systems and is extensively implemented. Despite the absence of easily exploitable logical vulnerabilities, attackers frequently manage to compromise Secure Boot through techniques like Fault Injections. [28]. Identifying these vulnerabilities can be challenging since they are often not apparent in the source code and require manual inspection of the disassembled binary code, examining each instruction individually. The main focus of this piece of research is FiSim [24] (Fault injection Simulator), developed by Riscure which is a training tool to test secure boot mechanism against Fault Injection Attacks. It is a ISA-level fault injection attack simulation tool. ISA-level simulation refers to the form of computer simulation that replicates the actions of a processor based on its instruction set architecture (ISA). At present, FiSim has the capability to assess software designed for ARM architectures, providing users with the ability to input a platform model defining address space, memory regions, and stack information. Based on the Unicorn framework and Capstone disassembler, it has a simple GUI for Windows platform only through which it is possible to simulate a fault injection attack and observe if it becomes successful or not. If an attacker has access to the binary file of their target, they can employ such a simulator to identify the most suitable location for executing a glitch attack. Conversely, system developers can utilize such a tool to evaluate the efficiency of their countermeasures against specific fault attacks.

FiSim was selected to extend for this piece of research because it was created using existing open-source components and unlike some other simulators it does not require the detailed model of the underlying hardware. It was developed with the idea of speed over accuracy. However, it also has many practical limitations and was mainly developed for showcasing faults on one secure boot implementation. In this next chapter, we discuss the working of FiSim, its architectural details, along with its limitations and the possible improvements related to this study.

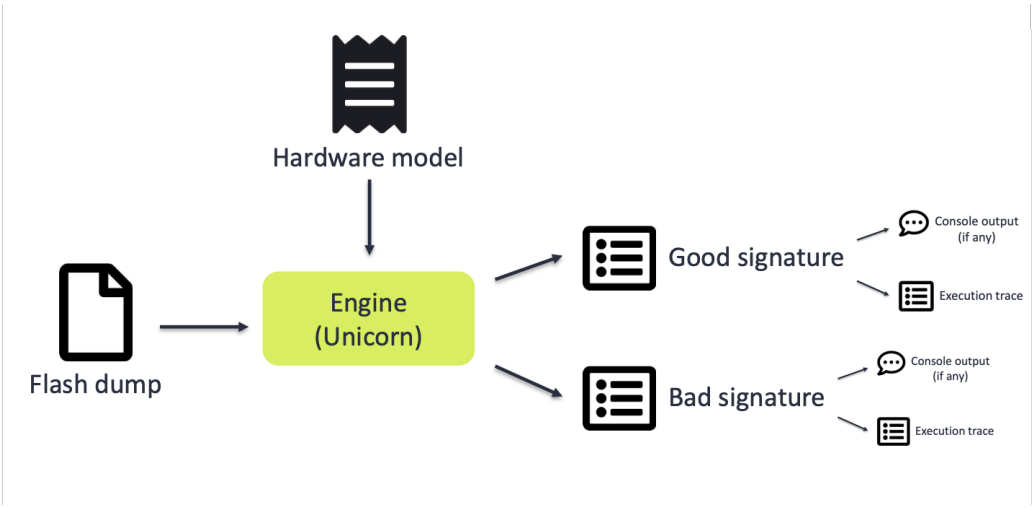


Figure 2.1: Simulating a binary in FiSim

2.1 Simulating a Binary with FiSim

By employing this simulator, it was possible to pinpoint specific locations within the binary of one secure boot implementation target where a successful glitch could potentially compromise the system's security. For instance, if a glitch is executed successfully, it could bypass the authentication of the subsequent boot stage or enable arbitrary code execution within the boot process. As a consequence, this would expose the cryptographic keys utilized for system protection or grant access to additional information crucial for devising a more scalable attack that doesn't rely on fault injection.

In order to glitch a secure boot process using FiSim, it needs to be provided with the hardware model of the device that we want to simulate for and the binary of the compiled source code or the dump of the flash which can be graphically seen in figure 2.1. The simulation engine runs it twice, once with a correct signature and the second time with the wrong signature and generates execution traces from both of them and console outputs. These varying signatures are used to run the secure boot process and decide if the next boot step has the correct signature or not. Then this execution trace is run in the fault generator and the simulation is run with all the instructions in the execution trace. The generator applies the faults on to the instruction.

If the code is vulnerable, with the flash dump and with the bad signature, it will be seen that the authentication succeeded and it starts to execute the next boot step. Hence it will be known that this particular glitch has bypassed the authentication.

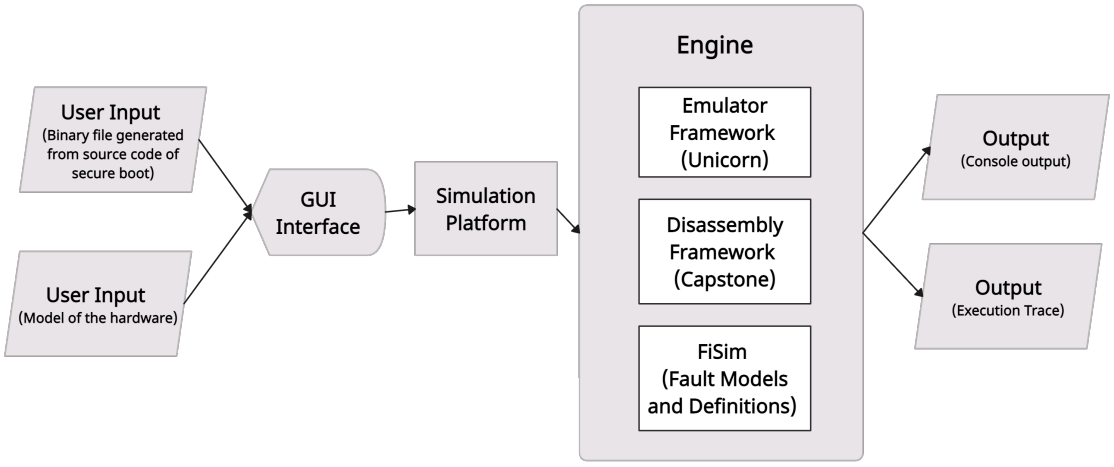


Figure 2.2: FiSim Architecture

In some of these cases it might be seen that the authentication was bypassed and it ends up executing the next boot step even though it was provided with an incorrect signature. So the secure boot mechanism is successfully bypassed.

2.2 Architecture

In order to enhance the capabilities of FiSim, it was mandatory to break down the architectural details of the simulator. The simulator is written in C# language but also using some functionalities of F# within. The overview of the architecture of FiSim can be seen in figure 2.2. As mentioned in the above section, the binary of the source code and the hardware model are provided as input and the simulator can be interacted with through the GUI interface. The model of the hardware is provided to the engine in order to know where to start executing from, where the stack is located, where is the heap, where is the code, where is the next boot step, where is the flash. Hence all these are required to define the address space of the system that is used by the software. These are then provided to the main Engine which contains the following three components.

Unicorn Emulator

Unicorn is a lightweight CPU emulator framework which provides support for multiple platforms and multiple architectures. It is based on the famous QEMU emulator but does not contains all the functionalities unlike QEMU. It can be called a simple instruction emulator where the code is feeded to the engine and it executes it. In Unicorn, the user has to define the setup for the emulation along with the rules of how the binaries should be emulated. FiSim simulator uses all these capabilities of the Unicorn engine to emulate binaries based on the hardware model that is provided as input.

Capstone Engine

FiSim uses the .NET implementation of Capstone disassembly framework to reverse engineer the binaries. Capstone is used to perform binary analysis and translate the machine code in to the assembly code. It also has a lightweight implementation and support for multiple architectures and platforms. Using this engine, the input binaries are translated into assembly instructions for emulation.

Fault Definitions and Models

The third important component of the engine is the Fault Definitions and Fault Models. FiSim defines two families of faults.

1. Transient Instruction Faults: These are the faults that might occur when an instruction is executed.

2. Cached Instruction Faults: These are the faults that might be persistent for the whole duration of a simulation.

Whereas a fault model is the simplified or abstracted representation of physical fault effects. The fault models in FiSim primarily focus on faults that can be represented by modifying the executed instructions during the simulation process. This means that FiSim is capable of altering the instructions being executed in order to simulate different fault scenarios. Additionally, it also supports fault models that involve setting one or more breakpoints on specific instructions, which allows for changing the state of the simulation at those breakpoints. FiSim implements two fault models for the secure boot process.

1. Transient Nop Instruction Fault Model: This model specifically focuses on

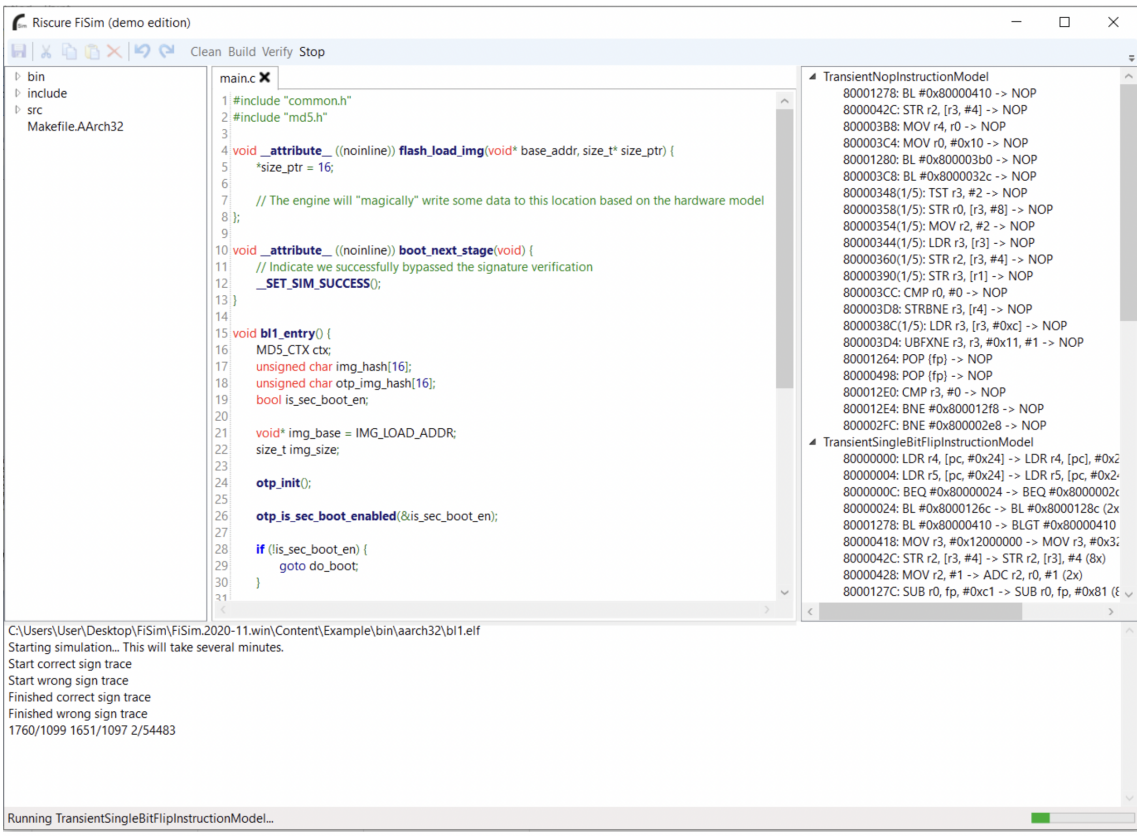


Figure 2.3: FiSim GUI Interface

transient faults introduced by inserting NOP (No Operation) instructions into the execution stream.

2. Transient Single Bit Flip Instruction Fault Model: In this model, a single bit in an instruction is flipped temporarily to simulate a transient fault.

GUI Interface

FiSim has a simple GUI interface which is only for Windows operating system currently as shown in figure 2.3. The main central window opens the file whose binary is being simulated by the engine, the bottom of the window shows the console output and the right window shows the execution of the fault models an displays all the instructions which can result in a potential glitch. Clicking on any particular instruction, corresponds to that line the c code, specifying the part of the code which can be vulnerable to fault injection attacks.

2.3 Limitations

Due to its minimized implementation and limited use cases, FiSim has a number of limitations which were found during this research. The most notable ones among them are:

1. It only contains the arm-none-eabi-gcc toolchain which provides support for only 32 bit ARM architecture binaries to be simulated.

2. The current implementation is only meant to test one bootloader, so it only has the fault model implementational support for bootloader binaries and to check if the boot process can go to the next boot stage if it is provided with some glitches.

3. Only implements two simplified transient fault models being instruction skipping faults and bit flip faults.

4. The details of the hardware model have to be added statically for every binary. Providing memory addresses to map code memory, stack memory, heap memory etc. for every binary file is a tiresome task if many binaries are to be tested.

5. FiSim systematically tests all potential faults by iterating through them using one or more fault models. It achieves this by repeatedly executing the target code while introducing each possible fault individually. However, it is important to note that not all simulated faults may be equally realistic or probable in a real device. Additionally, certain faults may not be physically possible to occur at all. This makes the simulation very slow and hard to implement other complex fault models.

6. Unicorn is a lightweight emulator but also has many constraints within its implementation.

In the next chapter, we will explain the approaches took in this research to address these limitations of FiSim.

Chapter 3

METHODOLOGY FOR MODIFICATIONS IN FISIM

As discussed in the previous chapter, there exist many limitations in FiSim, the most significant one of them being the fact that it only had support to simulate the binaries generated from boot-loader programs. The purpose of this research is to be able to add the support for other compiled binaries as well, mainly cryptographic algorithms. Fault injection attacks can have profound effects on cryptographic algorithms, undermining their security and compromising sensitive data. By intentionally injecting faults into the execution of cryptographic algorithms, attackers can exploit vulnerabilities and weaken the integrity, confidentiality, and authenticity of the cryptographic operations. These attacks can manifest in various ways. For example, injecting faults during encryption or decryption can result in the production of incorrect cipher-text or plain-text, leading to the leakage of sensitive information. Faults injected during key generation or key exchange can weaken the strength of the cryptographic keys, making them easier to guess or break. So we intend to modify FiSim to have support to simulate binaries compiled from cryptographic programs written in c language. Below we present the details about the modifications made in FiSim for this research to make applicable to more real world use cases and to make it able to support more kinds of programs other than secure boot.

3.1 Support for 64-bit ARM binaries

FiSim provided support for simulating only 32-bit ARM binaries as it only possessed arm-none-eabi tool-chain. A tool-chain is a collection of software development tools that are used together to build and compile software for the Linux operating system typically consisting of several components, including a compiler, linker, assembler etc. To add a better cross compilation support in FiSim++, we added aarch64-linux-gnu tool-chain. Along with some extension methods using the Unicorn engine, which make it able to compile ARM 64 bit binaries as well. However, ARM 32 bit binaries are widely used in most micro-controllers, such as the famous Cortex-M family, and are more relevant in terms of cryptographic devices. They are used in most real world use cases because of their energy efficiency, performance, and extensive ecosystem. Hence we did most of our testing using ARM 32 bit binaries in FiSim++.

3.2 Fault Models for Cryptographic algorithms

For this research, we extended the two existing fault models in FiSim and added their implementation for cryptographic algorithms as well.

For cryptographic algorithms, fault injections are considered a very crucial attack, as a single fault can during encryption can reveal the cipher's secret key, if the attack is well curated. Specific components of the cryptographic algorithm can be targeted by fault injection techniques, to extract sensitive data, such as secret keys. An attacker may take advantage of flaws during the algorithm's execution to recover the secret key and undermine the encryption or authentication process' security.

Fault Model: Skipping individual instructions

The first one being, skipping of individual instructions. In this method, specific instructions are purposefully skipped over or left out while a program is running. It examines how a system responds when some instructions are not carried out as intended.

In FiSim++, we iterate over the disassembled assembly instructions from the provided binary and replace individual instructions with a NOP (No Operation) so that a particular instruction is skipped. And then the original execution trace is compared with the new execution traces. For FiSim, a glitch is considered successful if the execution trace differs from the correct execution trace. In FiSim++, we follow the same approach and hence if the execution traces are different, it is declared that the skipping of that particular instruction can generate a fault in the system.

However, some instructions can have more notable repercussions than others. For example compare instructions are crucial, and if skipped, they can bypass major security checks in the program. Another example are the Branch or Jump instructions, if skipped, can change the execution flow of the program. Skipping them can lead to incorrect control flow and result in crashes. Memory read and write instructions are also susceptible to fault injections as skipping them can result in inconsistent data values.

Figure 3.1 shows how we can achieve this instruction skipping in real scenarios using clock glitches. Since the instructions are executed in a pipeline based on the rising edge, it's possible to skip an instruction by reducing the execution time. And this fault model replicates the same behaviour.

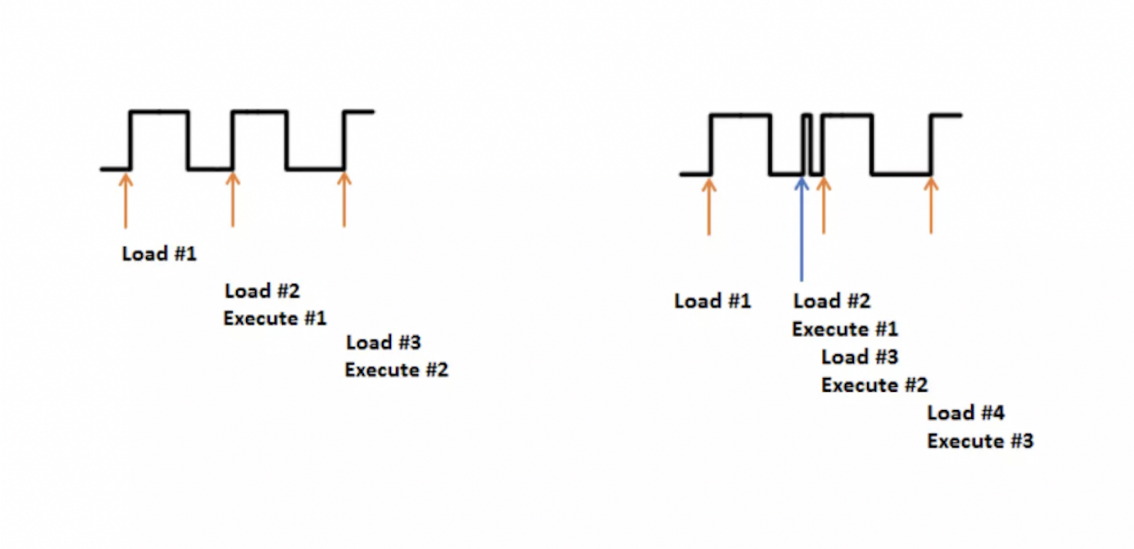


Figure 3.1: Instruction Skip through clock glitching

Fault Model: Flipping a bit in the instruction encoding

Flipping a bit in the instruction encoding fault model refers to the introduction of a fault through the alteration of one or more bits in an instruction's binary representation. This approach is very well known and most of the fault attacks on neural networks [18] [25] employ this specific model of fault injections. This fault model focuses on the commands being executed by a processor or microcontroller. And a simple example of it can be seen in figure 3.2. The instructions are converted into their binary representation, which consists of a series of 0s and 1s, when a program is compiled or assembled. For the processor, each bit in the instruction encoding corresponds to a particular operation, operand, or control signal and flipping a bit in this can have a variety of effects when the faulty instruction is executed.

```
MOV R0, R1        11100001101000000000000000000000
MOV R0, R2        11100001101000000000000000000010
```

Figure 3.2: Bit Flip in Instructions

1. Change in Operation: Flipping a bit in the instruction encoding can change the operation code (opcode) of the instruction, causing the processor to interpret it as a completely different instruction. Unexpected behavior or the execution of unintended operations may result from this.

2. Altered Operand: The value of a register or memory address being operated on can be altered by flipping a bit in an instruction's operand field. This could lead to the instruction manipulating different data than what was intended, which could lead to inaccurate computation or data corruption.

3. Control Flow Modification: By flipping a bit in a branch or jump instruction, you can change where in memory the program's control flow is directed. Inadvertent code execution, potential security check-bypassing, or program crashes may result from this.

In FiSim++, we focus on the flipping of single bits only. Which is done by iterating over all the instructions from the execution trace of a program, and performs a bit flip by using the bitwise XOR operation in the data and then applies the same approach to determine if the bit flip in that instruction causes a successful glitch or not.

3.3 Modelling the hardware for Different Binaries

Since it was developed to cater to only boot-loader programs, FiSim had a very particular hardware model defined, which was not able to simulate other binaries of c programs or cryptographic algorithms. Hence, in FiSim++ we added a generic hardware model which defines a memory region for code, a memory region for stack and for heap. And the memory of reasonable size to simulate many ARM 32 bit or ARM 64 bit binaries compiled from multiple examples of c programs and some cryptographic algorithms which will be discussed in the next chapter.
In the code snippet given in the listing 3.1, we see an example of defining a hardware model for an ARM 32 bit binary. We define the platform as AArch32, specifying the 32 bit ARM architecture. Additionaly, we define the number of maximum instructions for the simulator along with the memory regions and their size and addresses for our code, stack etc. The memory region defines the data, the size and the permissions given to that memory region. This object is used to model the hardware in order to simulate the binaries.

```
1 MyConfig simConfig = new MyConfig
2                 {
3                     Platform = Architecture.AArch32,
4                     EntryPoint = binInfo.Symbols["_start"].Address
   ,
5                     StackBase = 0x108000,
6                     MaxInstructions = 2000000000,
7                     AddressSpace = new AddressSpace {
```

```
8                   // Code
9                   { 0x8000, new MemoryRegion { Data = File.
  ReadAllBytes(binFilePath), Size = 0x100000, Permission =
  MemoryPermission.RWX } },
10                  // Stack
11                  { 0x108000, new MemoryRegion { Size = 0x100000
  , Permission = MemoryPermission.RW } },
12                  { 0x208000, new MemoryRegion { Size = 0x10000,
   Permission = MemoryPermission.RW  }  }
13              },
14              };
```

Listing 3.1: Creating a hardware model in FiSim++

3.4 Generating a PDF of the simulation results

In order to make FiSim++ more useful, we added the functionality to be able to generate a pdf file after the simulation of any binary is completed which contains the results of the simulation and the details about successful faults. This report can be used in future to refer to the simulation of that algorithm without running it again.

Chapter 4

TESTED ALGORITHMS

As mentioned in the previous chapters, that FiSim++ is intended to perform fault injection for cryptographic algorithms, so to test the simulator and its working we run it with some cryptography algorithms whose details and results are explained below.

4.1 AES-256

The first algorithm that we tested was the 256 bit version of the Advanced Encryption Algorithm (AES-256) [23], which is a symmetric encryption algorithm and is widely regarded as the standard symmetric encryption algorithm. And the key size that we use is 256 bits [15]. It is based on the principle of substitution-permutation network and uses 14 rounds for 256-bit keys.

For AES, a Differential Fault Analysis (DFA) is known to be able to recover the ciper key [26]. DFA is a type of active side-channel attack used in cryptanalysis. In order to reveal the internal states of cryptographic operations, the principle is to introduce faults unexpected environmental conditions into those operations [10]. By introducing single bit or single-byte faults during the computation, these attacks try to take advantage of the state's byte-wise processing. These attacks exploit the regular structure of the AES key schedule to infer bytes of the key by corrupting one or more bytes during the expansion of the last round key bits. [4].

Figure 4.1 shows the results generated by the simulation of the AES-256 bit by FiSim++. It can be seen that for the given algorithm, FiSim++ was able to run 2415 instruction iterations and among those, was able to find 610 successful glitches by applying the two fault models. The instructions that were responsible for a successful glitch are also given in the generated PDF. For the first fault model, it shows all the instructions, that if replaced with a NOP, can cause a successful glitch in the program. And for the second fault model it shows all the instructions, in which a single bit flip can change either the operator or the operand of the instruction and can result in a successful glitch. We can also add all the instructions which are not affected by the fault models and have a halt value false for them, but for the simplification purposes, we have avoided them for now. The report can also include

the corresponding lines of c code related to the glitch and can be used for security hardening of the program.

The simulation took almost 48 hours, as the execution had to be compared with all possible glitch executions in order to find a successful glitch.

4.2 Triple DES

The second algorithm that we tested using FiSim++ was triple DES, which also is a symmetric key block cipher encryption algorithm. It applies three rounds of Data Encryption Standard (DES) encryption to each data block for enhanced security and has a key size of 168 bits, as we use the 3-key version of Triple DES. [9]

As for DES, DFA is still a relevant approach by injecting faults into the encryption process and analyzing the resulting ciphertexts to reveal the secret key. And injecting faults in the middle rounds of the Triple DES algorithm has been found to be more effective than targeting the last two rounds [20].

The attacker can get pairs of ciphertexts one from an incorrect encryption and the other from one with intentional fault injections by doing so. The fundamental finding of DFA is that variations in the cipher texts can reveal important details about the secret key. After that, the attacker examines these fifferences to determine key bits.

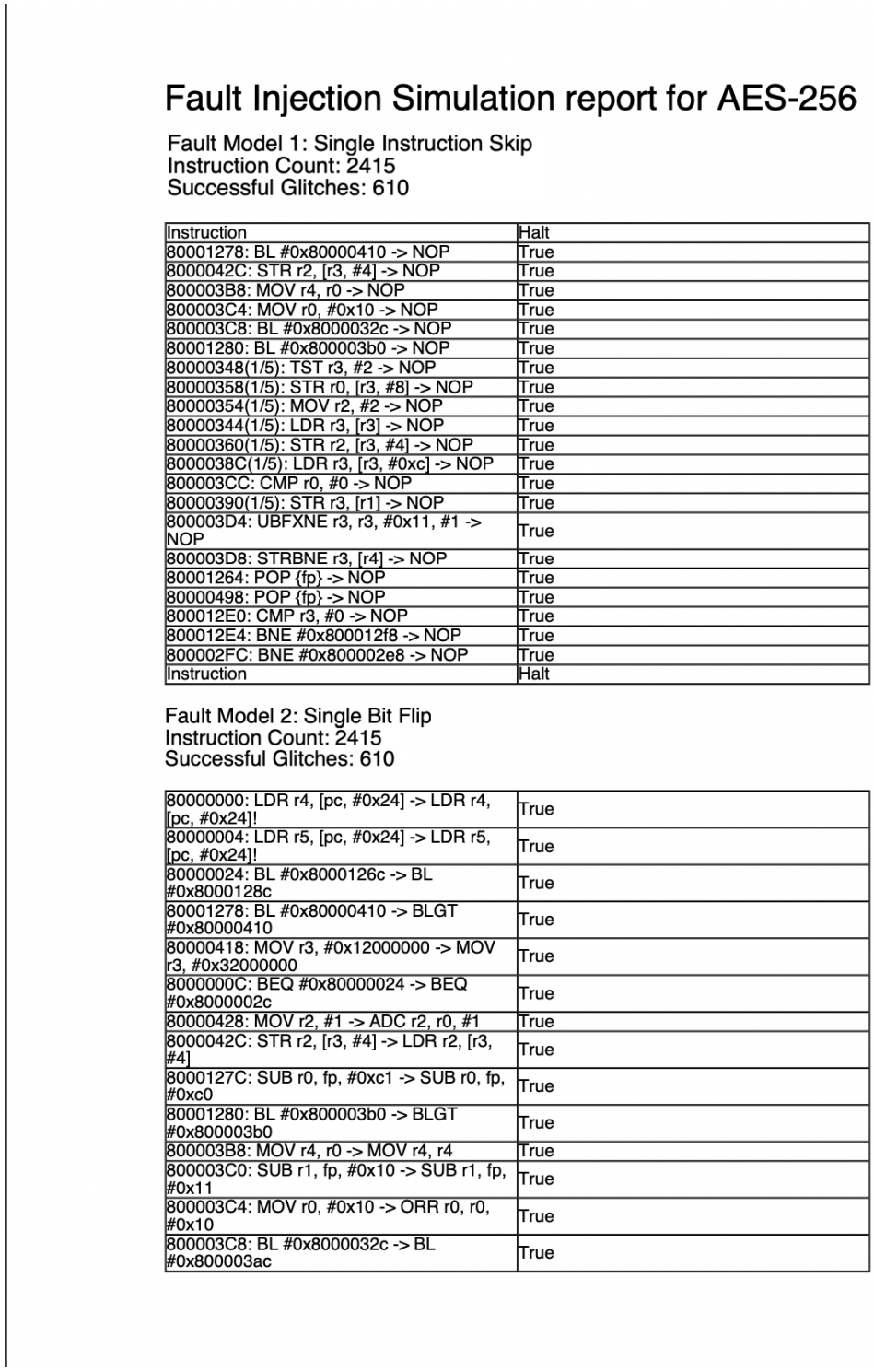
Countermeasures can be used during the encryption process to protect DES from DFA attacks. These defenses include mechanisms for fault detection and prevention, error-correcting codes, redundancy checks, and secure implementation methods. Hence, FiSim++ can be used for fault detection approach.

For Triple DES, FiSim++ was able to run 1535 different instructions and find 363 successful glitches. We generated a similar PDF for Triple DES as well, to pin point the areas which need security hardening for the given implementation of the algorithm and the simulation took almost the same amount of time as it did for the AES algorithm.

4.3 SHA256

The SHA-256 [3] hash function is a part of the SHA-2 hash family, which NIST standardized in 2002 as the replacement for SHA-1. It is a cryptographic hash function that accepts an input message and generates a fixed-size output of 256 bits.

Hash functions like SHA-256 are crucial in cryptography for a number of reasons. They ensure data integrity by producing a distinct hash value for each input, making



Fault Injection Simulation report for AES-256

Fault Model 1: Single Instruction Skip
Instruction Count: 2415
Successful Glitches: 610

Instruction	Halt
80001278: BL #0x80000410 -> NOP	True
8000042C: STR r2, [r3, #4] -> NOP	True
800003B8: MOV r4, r0 -> NOP	True
800003C4: MOV r0, #0x10 -> NOP	True
800003C8: BL #0x8000032c -> NOP	True
80001280: BL #0x800003b0 -> NOP	True
80000348(1/5): TST r3, #2 -> NOP	True
80000358(1/5): STR r0, [r3, #8] -> NOP	True
80000354(1/5): MOV r2, #2 -> NOP	True
80000344(1/5): LDR r3, [r3] -> NOP	True
80000360(1/5): STR r2, [r3, #4] -> NOP	True
8000038C(1/5): LDR r3, [r3, #0xc] -> NOP	True
800003CC: CMP r0, #0 -> NOP	True
80000390(1/5): STR r3, [r1] -> NOP	True
800003D4: UBFXNE r3, r3, #0x11, #1 -> NOP	True
800003D8: STRBNE r3, [r4] -> NOP	True
80001264: POP {fp} -> NOP	True
80000498: POP {fp} -> NOP	True
800012E0: CMP r3, #0 -> NOP	True
800012E4: BNE #0x800012f8 -> NOP	True
800002FC: BNE #0x800002e8 -> NOP	True
Instruction	Halt

Fault Model 2: Single Bit Flip
Instruction Count: 2415
Successful Glitches: 610

80000000: LDR r4, [pc, #0x24] -> LDR r4, [pc, #0x24]!	True
80000004: LDR r5, [pc, #0x24] -> LDR r5, [pc, #0x24]!	True
80000024: BL #0x8000126c -> BL #0x8000128c	True
80001278: BL #0x80000410 -> BLGT #0x80000410	True
80000418: MOV r3, #0x12000000 -> MOV r3, #0x32000000	True
8000000C: BEQ #0x80000024 -> BEQ #0x8000002c	True
80000428: MOV r2, #1 -> ADC r2, r0, #1	True
8000042C: STR r2, [r3, #4] -> LDR r2, [r3, #4]	True
8000127C: SUB r0, fp, #0xc1 -> SUB r0, fp, #0xc0	True
80001280: BL #0x800003b0 -> BLGT #0x800003b0	True
800003B8: MOV r4, r0 -> MOV r4, r4	True
800003C0: SUB r1, fp, #0x10 -> SUB r1, fp, #0x11	True
800003C4: MOV r0, #0x10 -> ORR r0, r0, #0x10	True
800003C8: BL #0x8000032c -> BL #0x800003ac	True

Figure 4.1: Output PDF generated for AES-256 algorithm

it simple to spot any changes or data tampering. In addition, digital signatures, key derivation, and password storage all use hashing algorithms.

In the case of SHA-256, DFA can be applied to the compression function, which receives a 512-bit message block and the current chaining value as inputs and outputs a new chaining value. In order to calculate the hash of any message, the attack aims to recover the internal state of the compression function. [12]

For SHA-256 algorithm, FiSim++ was able to run 986 different instructions and find 445 successful glitches. We generated a similar PDF for SHA-256 as well, to pin point the areas which need security hardening for the given implementation of the algorithm but the simulation was relatively faster and took no more than 12 hours to complete.

FiSim++ can be used to test many more cryptographic algorithms in the same way and generate successful glitches and find out areas that need to be considered for security. However, due to the long time taken to perform a fault injection simulation, we were only able to simulate a few algorithms as explained above.

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

To conclude this research, we see that fault injection attacks have emerged as a very crucial area of study and research in the field of Cybersecurity and have been given significant attention to identify and safeguard against fault injection attacks. And the development of various fault injection simulators have provided a valuable tool for evaluating system resilience against fault injection attacks, recognizing the vulnerable areas in the system and finding ways to harden the security of it. Among the many state-of-the-art fault simulators, we studied some of them for this research and their working.

Through this project, we have examined the concept of fault injection attacks, which involve intentionally introducing some kind of faults into a system to compromise its integrity or exploit vulnerabilities. For this purpose, we used an open-source fault injection simulator, FiSim++, which would previously be only used to test boot-loaders and secure boot programs. But during the course of this research, we modified the functionalities of FiSim and added some additional features in it to be able to simulate cryptographic algorithms and find vulnerabilities in cryptographic programs. FiSim++ works with binaries compiled in ARM 32 or 64 bit architecture and uses those binaries to inject faults in the respective disassembled programs. We tested it some simple C programs first and then with some implementation of cryptographic algorithm such as AES, TripleDES, SHA256 etc. The simulation of these algorithms takes some time but ultimately we were able to achieve the results of the injected faults in the algorithms and generate a report based upon it, defining the areas which can be vulnerable to fault injections in real-world scenarios, which need to be considered for security patches. The ultimate goal is to make FiSim++ able to simulate post-quantum crypto algorithms and inject faults in them.

The extended code of this work would be uploaded on a public GitHub repository for a widespread use.

In conclusion, we believe that such a tool can be used in multiple applications and real-world scenarios and it can be able to harden the security of microcontrollers, and cryptographic systems. Its easy interface and support for multiple architectures

and programs can make it a valuable asset for security researchers and security testers.

5.2 Contributions

In table 5.1, we present a comparison of FiSim with FiSim++ and summarize our contributions to this project. We see that FiSim++ has major advantages over FiSim in terms of the targeted architectures, supported program binaries and the generated output formats. However, it still supports the same fault models and has the same criteria for defining a successful fault. We see that there is a significant decrease in the speed of the simulation, as it depends upon the complexity of the algorithm. The bootloader program tested by FiSim was fairly simple as compared to the cryptographic algorithm, hence, the time taken to simulate the binaries increases.

5.3 Future Work

While this report has provided an overview of fault injection attacks, the development of fault simulators and the particularly the functioning of FiSim++, there are still several avenues for future research and exploration on this project which could not be considered for the timeline of this research. The ideas for future work include:

1. One very promising aspect for future work is the utilization of FiSim++ to evaluate the robustness and security of post-quantum cryptographic algorithms. With the rise of quantum computing, traditional cryptographic algorithms face the risk of being rendered obsolete, potentially compromising the security of sensitive information. Post-quantum cryptographic algorithms, on the other hand, aim to provide resilience against quantum attacks. The selected group of encryption algorithms by The U.S. Department of Commerce's National Institute of Standards and Technology (NIST) [21], which have been specifically designed to withstand the potential threat posed by future quantum computers are CRYSTALS-Kyber [2], CRYSTALS-Dilithium [19], FALCON [11] and SPHINCS+ [6]. And we want them to be tested through FiSim++ and generate the fault injection reports. This approach would not only contribute to the development of more secure post-quantum cryptographic schemes but also aid in establishing a comprehensive understanding of the challenges and requirements associated with integrating such algorithms into real-world systems.

2. We want FiSim++ to have the functionality to be able to select a particular

Criteria	**FiSim**	**FiSim++**
Target Architecture	ARM 32	ARM 32 and ARM 64
Supported Binaries	One bootloader program (Pineboot)	Any, tested on 3 crypto algorithms
Supported Fault Models	Single Instruction skipping fault model and Single bit flip fault model	Single Instruction skipping fault model and Single bit flip fault model
Generated Output Format	Console output	Console output along with a generated PDF
Criteria for Successful glitches	Comparison of generated execution traces with correct execution trace	Comparison of generated execution traces with correct execution trace
Speed of the simulation	40.42462 instructions/second	0.01402 instructions/second (Depends upon the input algorithm)

Table 5.1: Comparison between FiSim and FiSim++

section from the c code and then inject faults in that part of the disassembly instead of the whole program. This way we can speed up the process of simulation and also add more flexibility for the user to select a vulnerable part of the code and test it with the simulator.

3. Currently, FiSim++ only employs two techniques of fault injections, called fault models in our context. The two being, instruction skipping and bit flipping. These two techniques cover a wide range of glitches in terms of

real-world injections, however, there are many more techniques which can be added in FiSim++ to cover more use cases related to actual glitches. Some of the potential fault models [7] that can be added are:

- Bit set/reset: In this fault model, the attacker has the ability to selectively change a bit's value, either setting it to "1" or "0." For blind fault attacks, where the attacker takes use of the changed bit values to obtain unauthorized access to or manipulate data, this potent fault model can be used.
- Random byte: This failure model introduces a less precise defect in which the value of a particular byte is changed arbitrarily to another value. Although it is regarded as a relaxed fault model, Differential Fault Analysis (DFA) attacks can still be successful with this fault model. An attacker can take advantage of weaknesses in cryptographic methods and obtain unauthorized information by causing random byte errors.
- Execution faults: This fault model applies to faults in Field-Programmable Gate Arrays (FPGAs), where setup errors have an impact on the values being processed. Attackers can use execution failures to target physically unclonable functions that produce unique identifiers. Attackers can jeopardize the security of these functions and potentially get around access controls by inserting defects.
- Stuck-at faults: Faults that get stuck permanently change the value of recorded data, substituting a different value. The Stuck-At Fault Attack (SIFA) and other attacks frequently employ this fault model. Additionally, the generation of biased numbers can be caused by the introduction of stuck-at errors in true random number generators (TRNGs). These biases can be used by attackers to launch cryptographic attacks.
- Multiple bit flip: Multiple bits within a system or piece of data are simultaneously flipped or manipulated in the multiple bit flip fault model, a fault injection technique. Compared to single bit flip models, this fault model is more potent and aggressive since it has the potential to cause more widespread disruptions and evade error-correction mechanisms.

4. The current version of FiSim++ supports the ARM architecture only, which covers most of the widely used Cortex-M micro controllers. But to make it

more apt for other use cases too, we can add support for other architectures such as x86, IA-64 etc. by adding their tool-chain extensions.

5. The main concern for many fault simulators is speed to simulate binaries and calculate meaningful results in certain amount of time. But fault injection is an iterative process that involves repeating the experiment multiple times to gather sufficient data and validate results, hence the operation of FiSim++ for cryptographic algorithms is a time consuming process as well. So optimizing the time taken by the simulation is also a future work area that needs attention.

Chapter 6

ENVIRONMENT AND LEARNING OUTCOMES

6.1 Environment of the Internship

This research was carried out at the Cryptographic Engineering and Side-Channel Analysis (CESCA) Lab at the Radboud University which is working actively to provide secure implementations of cryptographic systems in embedded systems. The group has ongoing researches on multiple topic related to the above mission such as, physical attacks on embedded crypto devices, secure cryptographic implementations, FPGA security, AI and security, lightweight cryptography, leakage simulators etc. The researchers and professors at the Lab are responsible for many revolutionary and cutting-edge researches in the field of cryptography and side channel attacks. The group would have a lunch talk every friday where some new research by someone from the group or an external researcher would be presented. I attended presentations related to neural networks in side channel attacks, cryptographic developments and many other topics that were very insightful for me and gave me an overview of many new findings in the field of cybersecurity. I would have weekly meetings with my supervisor, to discuss the updates and ideas about the research topic which were very helpful in understanding the subject matter and the approach to pursue it.

6.2 Educational and Learning Outcomes

Working with such a group on FiSim++ opened a lot of learning avenues for me including:

1. A thorough understanding of fault injection attacks, their implementation, and the multiple ways they can be carried out.
2. State-of-the-art fault injection simulators and techniques.
3. The art of simulation: How to simulate fault injection attacks without using actual hardware boards and chips. Simulation is a very powerful tool and has many applications in the field of Cybersecurity.
4. Understanding of the Differential Fault Analysis (DFA) techniques and how they can be used to attack cryptographic algorithms.

5. An introduction to post-quantum crypto algorithms, as some people in the group are working on post-quantum crypto algorithms so I was introduced to some of them such as Kyber, Dilithium etc.

6. Assembling and disassembling of machine and c code.

7. Working with binary files and being able to cross compile binaries for multiple architectures and platforms.

8. Knowledge of Vulnerability Assessment: Through fault injection simulation, I learned how to assess the vulnerability of systems or software to different types of faults.

9. Research Skills: Working in a research lab allowed me to develop essential research skills, such as literature review, experimental design, data collection, analysis, and interpretation.

BIBLIOGRAPHY

[1] Asmita Adhikary and Ileana Buhan. "SoK: Assisted Fault Simulation-Existing Challenges and Opportunities Offered by AI". In: *Cryptology ePrint Archive* (2022).

[2] Roberto Avanzi et al. "Crystals-kyber". In: *NIST, Tech. Rep* (2017).

[3] B-Con. *B-Con/crypto-algorithms*. `https://github.com/B-Con/crypto-algorithms`. Year 2023.

[4] Alessandro Barenghi et al. "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures". In: *Proceedings of the IEEE* 100.11 (2012), pp. 3056–3076.

[5] Ali Basiri et al. "Chaos engineering". In: *IEEE Software* 33.3 (2016), pp. 35–41.

[6] Daniel J Bernstein et al. "The SPHINCS+ signature framework". In: *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 2019, pp. 2129–2146.

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

[8] Common Criteria. *Common Criteria Portal*. `https://commoncriteriaportal.org/cc/`. 2023.

[9] The Cryptography. *TripleDES.c*. `https://github.com/The-Cryptography/C/blob/master/ciphers/TripleDES.c`. Accessed: June 18th, 2023. 2023.

[10] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. "Differential fault analysis on AES". In: *Applied Cryptography and Network Security: First International Conference, ACNS 2003, Kunming, China, October 16-19, 2003. Proceedings 1*. Springer. 2003, pp. 293–306.

[11] Pierre-Alain Fouque et al. "Falcon: Fast-Fourier lattice-based compact signatures over NTRU". In: *Submission to the NIST's post-quantum cryptography standardization process* 36.5 (2018).

[12] Ronglin Hao et al. "Algebraic fault attack on the SHA-256 compression function". In: *International Journal of Research in Computer Science* 4.2 (2014), pp. 1–9.

[13] Florian Hauschild et al. "ARCHIE: A QEMU-Based Framework for Architecture-Independent Evaluation of Faults". In: *2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*. IEEE. 2021, pp. 20–30.

[14] Max Hoffmann, Falk Schellenberg, and Christof Paar. “Armory: Fully automated and exhaustive fault simulation on arm-m binaries”. In: *IEEE Transactions on Information Forensics and Security* 16 (2020), pp. 1058–1073.

[15] ilvn. *aes256*. `https://github.com/ilvn/aes256`. Accessed: June 19th, 2023. 2023.

[16] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. “FERRARI: A flexible software-based fault and error injection system”. In: *IEEE Transactions on computers* 44.2 (1995), pp. 248–260.

[17] Rakesh Kumar Lenka, Sarthak Padhi, and Kabita Manjari Nayak. “Fault injection techniques-a brief review”. In: *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*. IEEE. 2018, pp. 832–837.

[18] Yannan Liu et al. “Fault injection attack on deep neural network”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2017, pp. 131–138.

[19] Vadim Lyubashevsky et al. “Crystals-dilithium”. In: *Algorithm Specifications and Supporting Documentation* (2020).

[20] Xiangliang Ma et al. “Differential fault analysis on 3DES middle rounds based on error propagation”. In: *Chinese Journal of Electronics* 31.1 (2022), pp. 68–78.

[21] National Institute of Standards and Technology. *NIST Announces First Four Quantum-Resistant Cryptographic Algorithms*. 2022. URL: `https://www.nist.gov/news-events/news/2022/07/nist-announces-first-four-quantum-resistant-cryptographic-algorithms`.

[22] Vasileios Porpodas. “Zofi: Zero-overhead fault injection tool for fast transient fault coverage analysis”. In: *arXiv preprint arXiv:1906.09390* (2019).

[23] Vincent Rijmen and Joan Daemen. “Advanced encryption standard”. In: Proceedings. 1998.

[24] Riscure. *FiSim*. `https://github.com/Riscure/FiSim`. Accessed: June 18th, 2023. 2023.

[25] A Ruospo et al. “Assessing convolutional neural networks reliability through statistical fault injections”. In: *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2023, pp. 1–6.

[26] Dhiman Saha, Debdeep Mukhopadhyay, and Dipanwita RoyChowdhury. “A diagonal fault attack on the advanced encryption standard”. In: *Cryptology ePrint Archive* (2009).

[27] Rickard Svenningsson et al. “MODIFI: a MODel-implemented fault injection tool”. In: *Computer Safety, Reliability, and Security: 29th International Conference, SAFECOMP 2010, Vienna, Austria, September 14-17, 2010. Proceedings 29*. Springer. 2010, pp. 210–222.

[28] Niek Timmers, Albert Spruyt, and Marc Witteman. “Controlling PC on ARM Using Fault Injection”. In: *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 2016, pp. 25–35. DOI: 10.1109/FDTC.2016.18.