

Telecommunications engineers' training cycle

*Option :*

Cyber Security and Defense

## **Graduation project report**

*Topic :*

### **« Fault Simulation and Injection on Microprocessor Architectures »**

*Realized by :*

**Oumayma Teyeb**

*Supervisors:*

Prof. Fethi Tlili, Professor, Head of the GRESCOM Research Lab at SUP'COM,  
Tunisia

Ihab Alshaer, PhD student, Grenoble Alpes University, France

Prof. Vincent Beroulle, Head of LCIS Laboratory, Professor at UGA/Grenoble INP-  
Esisar

Dr.Christophe DELEUZE Grenoble INP - Esisar, UGA,

*Work proposed and realized within:*

The Laboratory for Design and Integration of Systems ( LCIS )



Année universitaire : 2021 - 2022

# foreword

This project is being carried out as part of a graduation internship at the Higher School of Communication of Tunis (SUPCOM) during which I had the chance to join the Laboratory for Design and Integration of Systems ( LCIS ).

# Acknowledgment

I would like to take this opportunity to express my heartfelt gratitude to all the people who helped me during this internship.

I am fortunate to have had the opportunity to work in the LCIS laboratory. A rich and highly dynamic learning environment provided me with valuable hands-on experience and a solid knowledge base for my current and future work.

In particular I want to thank my supervisor **Ihab Alshaer** for providing me with the guidance and help throughout these challenging months as well as the chance he provided to work on such a fascinating topic as part of his Phd thesis.

A special thanks goes to **Prof. Fethi Tlili**, head of GRES'COM laboratory, professor at Sup'com, for his unwavering support, motivational feedback, encouragement and for his valuable advice.

I would like to express my gratitude to **Prof. Vincent Beroulle**, head of LCIS Laboratory, professor at UGA/Grenoble INP- Esisar, for his assistance, generosity and his useful and constructive recommendations on this project.

I wish to extend my special thanks to **Caroline Palisse**, Administrative and Financial Assistant at LCIS for making my integration and my mobility easier.

Furthermore I would also like to acknowledge with much appreciation the crucial role of **Marie Vergriete**, Research engineer and computer manager at LCIS for her valuable technical support on this project.

Thanks to all of the **SUPCOM teachers** who were involved in my educational progress throughout my engineering studies.

I would like to express my gratitude to the members of the **jury** for the honor they have bestowed upon me by reviewing my graduation project, and I hope that they can find the clarification and inspiration that they seek in this report.

At the closure, I would like to express my heartfelt gratitude to **my family** for the constant cheer leading and moral support and for being the backbone to every single success in my life.

# Abstract

Given growing attack techniques and technology, the protective mission should be emphasized. Fault attack mitigation needs knowing the attack vector. This entails discovering, evaluating, and analyzing code defects that may be exploited. It also requires cost-effective software and hardware security at various levels.

This project aims to build a mechanism for autonomously simulating software models and analyzing fault injection effects.

First, it mimics realistic and effective models utilizing code and register state that abstracts the lowest fault injection levels. Second, it compares previous outcomes to physical injection and RTL fault simulation.

This study will assist developers in developing complicated and effective defenses by estimating which models are likely to run on the target board and how frequent they are.

---

**Keywords :** Hardware security, software fault simulation, fault injection, automated analysis...

---

# Résumé

Compte tenu des techniques et de la technologie d'attaque croissantes, la mission de protection doit être soulignée. L'atténuation des attaques par faute nécessite de connaître le vecteur d'attaque. Cela implique de découvrir, d'évaluer et d'analyser les défauts de code susceptibles d'être exploités. Cela nécessite également une sécurité logicielle et matérielle rentable à différents niveaux.

Ce projet vise à construire un mécanisme de simulation autonome de modèles logiciels et d'analyse des effets d'injection de fautes.

Tout d'abord, il imite des modèles réalistes et efficaces utilisant un état de code et de registre qui résume les niveaux d'injection de fautes les plus bas. Deuxièmement, il compare les résultats précédents à l'injection physique et à la simulation de fautes RTL.

Cette étude aidera les développeurs à développer des défenses efficaces en estimant quels modèles sont susceptibles de s'exécuter sur la carte cible et à quelle fréquence ils sont.

---

**Mots clés :** Sécurité matérielle, simulation de fautes logicielles, injection de fautes, analyse automatisée...

---

# Contents

<b>foreword</b>	<b>I</b>
<b>Acknowledgment</b>	<b>II</b>
<b>Abstract</b>	<b>III</b>
<b>Résumé</b>	<b>IV</b>
<b>General introduction</b>	<b>1</b>
<b>1 Fault injection and simulation on microprocessor architectures context</b>	<b>3</b>
1.1 Introduction	4
1.2 Project Context	4
1.3 Hosting Company	4
1.4 Project CLAM: Cross-Layer Fault Analysis for Microprocessor Architec- tures Overview	5
1.5 Our contribution to the project	6
1.6 Conclusion	7
<b>2 Background and related work</b>	<b>8</b>
2.1 Introduction	9
2.2 Methodologies and Terms	9
2.2.1 Fault	9
2.2.2 Fault model	9
2.2.3 Fault injection	9
2.2.4 Hardware-based fault injection	9
2.2.5 Software-based fault injection	10
2.3 Related work	10
2.4 Conclusion	12
<b>3 Software model simulation and analysis automation conception</b>	<b>13</b>
3.1 Introduction	14
3.2 Need Recall	14
3.3 Software model simulation and analysis automation goals	14
3.4 Software model simulation and analysis automation presentation	16
3.5 Software model simulator and analysis automator's functional architecture	17
3.6 Software model simulator and analysis automator's functional process	18
3.7 Conclusion	21

<b>4</b>	<b>Environment setup</b>	<b>22</b>
4.1	Introduction	23
4.2	Technologies and modules used	23
4.3	Assembly Functions	25
4.3.1	Micropython	25
4.3.2	Assembly syntax	26
4.4	Target board	27
4.5	Conclusion	29
<b>5</b>	<b>Software model simulation and analysis automation development</b>	<b>30</b>
5.1	Introduction	31
5.2	Software fault models	31
5.2.1	Instruction Skip	31
5.2.2	Instruction Skip and instruction Repeat	32
5.2.3	Double instruction corruption	33
5.2.4	Instruction skip, instruction repeat and double instruction corruption	35
5.2.5	Instruction skip and instruction corruption	37
5.2.6	Instruction skip and new instruction execution	37
5.2.7	Instruction corruption with zeros	38
5.2.8	One Operand Corruption	38
5.2.9	One instruction corruption, changing it to MOV instruction	39
5.3	Automation Module	40
5.4	Conclusion	41
<b>6</b>	<b>Fault injection and simulation on microprocessor architectures tests and results</b>	<b>42</b>
6.1	Introduction	43
6.2	Fault Model Simulation	43
6.2.1	Getting the Source Code	43
6.2.2	Installing the requirements	43
6.2.3	Running the program	44
6.2.4	Observed results	44
6.3	Physical fault injection	46
6.3.1	Clock glitch	46
6.3.2	ChipWhisperer	47
6.3.3	Target	47
6.3.4	Environment setup	48
6.3.5	Testing the normal behaviour	49
6.3.6	Clock glitch parameters	49
6.3.7	Attacking VerifyPin	51
6.3.8	Understanding the fault	52
6.4	Conclusion	54
	<b>General conclusion</b>	<b>56</b>

# List of Figures

1.1	LCIS logo. . . . .	5
1.2	CLAM Methodology [1]. . . . .	6
3.1	Software model simulation and analysis automation goal. . . . .	15
3.2	Software model simulator and analysis automator. . . . .	16
3.3	Software model simulator and analysis automator's functional architecture. . . . .	17
3.4	Software model simulator and analysis automator's functional process. . . . .	20
4.1	Python Logo. . . . .	23
4.2	Pandas Logo. . . . .	23
4.3	Numpy Logo. . . . .	24
4.4	Jupyter Logo. . . . .	24
4.5	FPDF Logo. . . . .	24
4.6	Capstone Logo. . . . .	25
4.7	Keystone Engine Logo. . . . .	25
4.8	Micropython Logo. . . . .	25
4.9	Micropython syntax for Thumb-2. . . . .	26
4.10	Instruction cycle. . . . .	27
4.11	Instruction cycle - Fetch Stage. . . . .	28
5.1	Skip fault model. . . . .	31
5.2	Skip and repeat fault model. . . . .	32
5.3	Double instruction corruption fault model - case 1. . . . .	34
5.4	Double instruction corruption fault model - case 2. . . . .	35
5.5	Instruction skip, instruction repeat and new instruction execution - case 1. . . . .	36
5.6	Instruction skip, instruction repeat and new instruction execution - case 2. . . . .	36
5.7	Instruction skip and instruction corruption fault model. . . . .	37
5.8	Instruction skip and new instruction execution fault model. . . . .	37
5.9	Instruction corruption with zeros - case 1. . . . .	38
5.10	Instruction corruption with zeros fault model - case 2. . . . .	38
5.11	One operand corruption fault model. . . . .	39
5.12	One instruction corruption, changing it to MOV instruction. . . . .	40
5.13	Fault injection outcomes CSV file. . . . .	40
6.1	Project folder's content. . . . .	43
6.2	Virtual environment creation. . . . .	44
6.3	Successful completion of the program. . . . .	44
6.4	Software fault simulation outcomes. . . . .	45
6.5	Generated report. . . . .	45



6.6	Clock glitch. . . . .	46
6.7	ChipWhisperer Logo. . . . .	47
6.8	Target code. . . . .	48
6.9	ChipWhisperer board setup. . . . .	48
6.10	Setting up and programming the target. . . . .	49
6.11	Testing the normal behaviour : Wrong password. . . . .	49
6.12	Testing the normal behaviour : correct password. . . . .	49
6.13	Characterization code. . . . .	50
6.14	Offset and width pair leading to a fault. . . . .	51
6.15	Attacking VerifyPin. . . . .	51
6.16	For Loop. . . . .	52
6.17	Target software - Assembly code. . . . .	54

# Liste des sigles et acronymes

<b>ALU</b>	<i>Arithmetic–Logic Unit,</i>
<b>CPU</b>	<i>Central Processing Unit</i>
<b>HWIFI</b>	<i>Hardware-implemented Fault Injection</i>
<b>ISA</b>	<i>Instruction Set Architecture</i>
<b>MAR</b>	<i>Memory Address Register</i>
<b>PC</b>	<i>Program Counter</i>
<b>PIN</b>	<i>Personal Identification Number</i>
<b>RTL</b>	<i>Register Transfer Level</i>
<b>SFI</b>	<i>Software Fault Injection</i>

# General introduction

With the rise of the IoT and the need for more connection, embedded systems have found new applications in many different fields. Which raises new concerns about the safety and security of such devices particularly considering the ease physical access to these gadgets.

Given the ever-evolving nature of assault methods and technology, the protection mission should be given even higher emphasis. One of the best-known security strategies is early mitigation, which entails designing without flaws and implementing without bugs. To achieve this level of assurance, researchers used a range of methodologies, including well-known fault injection testing.

Fault attack mitigation requires a detailed understanding of the faulty attack vector. To begin, this requires detecting, inspecting, and assessing the flaws that may result in exploitable code vulnerabilities. Furthermore, it involves the development of cost-effective hardware and software defenses at many levels: software and hardware. In fact, defining the issue at a single level of analysis results in either over engineering or under engineering of the countermeasures. By choosing a suitable fault model, the lowest levels of fault injection layers may be abstracted away at the expense of the modeling accuracy as well as the simulation speed.

Over designing the system with both spatial and temporal redundancy is the common method of protecting it against fault attacks. This is a costly solution that can only be used in configurations in which efficiency and/or costs take a back seat to security. Moreover, the undersigning would leave the system vulnerable to a variety of attacks that may be exploited, putting the system at danger. This technique is only appropriate for systems where security is not a significant job in comparison to performance. To begin the process of developing countermeasures that are more effective, it is vital to first have an understanding of how faults are introduced and propagated across a system.

This project presents an automated technique for software model simulation and outcome analysis. Taking into consideration the desired code and the initial state of registers, simulates the program at the ISA level and permits fault injections on user provided instructions. By simulating appropriate fault models, the lowest levels of fault injection layers may be abstracted. In the second step, it will compare the previously obtained findings with those of physical injection and RTL simulation. This study will assist in the construction of sophisticated and effective countermeasures, since the developer will be able to determine which models can be executed on the target board and how likely this is to occur.

The rest of the report is organized as follows: in the first chapter, “**Fault injection and simulation on microprocessor architectures context**”, the project will be put into its framework namely the host organization will be defined and the scope of the project will be detailed.

In the second chapter, “**Background and related work**”, important terms and terminologies will be defined before examining the associated industry work.

In the third chapter, “**Software model simulation and analysis automation conception**” the motivation behind the tool proposed will be recalled and its architecture and functional details will be explained.

The fourth chapter “**Environment Setup**” defines all of used modules and technologies and explains the developed assembly syntax. Before drawing a conclusion, the simulated target board will be described, together with the properties that identify it.

The fifth chapter “**Software model simulation and analysis automation development**” walks through the proposed tool’s development process, from software model simulation through the automatic fault injection analyzer module.

Before concluding the report, in the sixth chapter “**Fault injection and simulation on microprocessor architectures tests and results**”, the tool will be put into action. Fault injection output is sent to the software tool in order to witness the analysis and simulation of software models. And in the subsequent step, a real case scenario will be demonstrated by performing fault injections to bypass a password check and acquire unauthorized access.

# Chapter 1

## Fault injection and simulation on microprocessor architectures context

## 1.1 Introduction

Before delving into the intricacies of the project and offering in-depth insights about it, the project should first be organized into its framework. For this reason, the context of fault injection and simulation on microprocessor architectures project will be introduced in this chapter, as will the organism that offers it. This part comes to a close by outlining the problem that is being addressed, offering an overview of the project, and discussing how I have contributed to it.

## 1.2 Project Context

This project is being carried out as part of a graduation internship. Its goal is to foster independence and responsibility, foster group cohesion and foster a sense of pride in one's work, and, of course, provide an opportunity to put knowledge gained into practice. Problem-solving is only one of many transferable talents that may be picked up while working on a project. The scope of this work is in the realm of hardware security. This study is part of the project "CLAM" (Cross-Layer Fault Analysis for Microprocessor Architectures). It's a collaboration between three French labs with overlapping but distinct areas of expertise: the LCIS (in Valence), which focuses on hardware fault simulation at the RTL level and the creation of fault injection tools (voltage and clock glitch generators, Electromagnetic assault); the TIMA Laboratory (Grenoble), which has experience with fault assessment, modeling, and emulation; and the Verimag Laboratory (also in Grenoble), expert in software vulnerability investigation using static analysis techniques. All of these laboratories have created national and international waves with their fault injection studies.

## 1.3 Hosting Company

Hosted by a group from Grenoble-INP-affiliated University of Grenoble Alpes (UGA), the LCIS is a public research laboratory devoted to integration, design, and systems.

More than sixty professionals from the fields of computer science, automation, and electronics collaborate on research into embedded and communicative systems at the LCIS.



Figure 1.1: LCIS logo.

The lab's research extends to a wide variety of practical domains, including RFID, cyber-physical systems, networked ecosystems (both natural and artificial), and the Internet of Things that are distributed between three teams :

- **ORSYS: Optoelectronic and Radiofrequency Systems** : This team focus on improving methods for communicating, processing signals, and measuring distance using radio frequency and wireless technology.
- **CO4SYS : Coordination, cooperation, control of complex systems** : This team models, controls, and supervises complicated artificial systems composed of several interacting units. The team utilizes automation and computer science methodologies to represent these linked but loosely coupled node systems.
- **CTSYS : Safety and security of embedded and distributed systems** : Embedded and distributed system security is an area of focus for the CTSYS team. Researchers from several fields (electronics, IT, telecom) investigate the various components of embedded systems, from hardware to application to insure their safety and security.

During this internship, I had the chance to join the CTSYS team in their Cross-Layer Fault Analysis for Microprocessor Architectures project.

## 1.4 Project CLAM: Cross-Layer Fault Analysis for Microprocessor Architectures Overview

Securing the Internet of Things market and other critical cyber physical infrastructures requires a comprehensive study of these components' vulnerabilities and the development of suitable hardware and software remedies.

Because of the increasing complexity of processors and the programs they run, existing software fault models used to investigate code vulnerabilities are no longer enough to describe the spectrum of erroneous behaviors in contemporary architectures. Architectural designers have increasingly added many complex hardware components (such as pipelines, cache memory, branch prediction, and speculative execution) onto processors in an attempt to boost program speed.

Simultaneously, fault injection technologies are constantly developing, allowing for higher-order injections (multi-temporal and multi-spatial). In the face of these attacks, designers want safeguards that can either discover or mask the implications of added flaws. Certain preventive measures may be implemented using either hardware (isolation methods, element duplication) or software (instruction/algorithm duplication, signature verification). A realistic software and/or hardware fault model is necessary to adequately create and test these defenses.

In order to facilitate the process of developing countermeasures, a number of research studies centered on one level of fault characterisation have been carried out. These levels include the Instruction-Set Architecture and the RTL level. On the other hand, since the fault model isn't accurate, this may cause the defenses to be either under- or over-engineered. In the first scenario, a security risk may still exist, and as a result, it may be exploitable; in the second scenario, this implies increasing expense without any justification, and it may also reduce performance.

The fault models that are created should be realistic and effective. As a result, the analysis will not be confined to one level, but will be comprised of three layers of investigation as demonstrated in the figure 1.2.

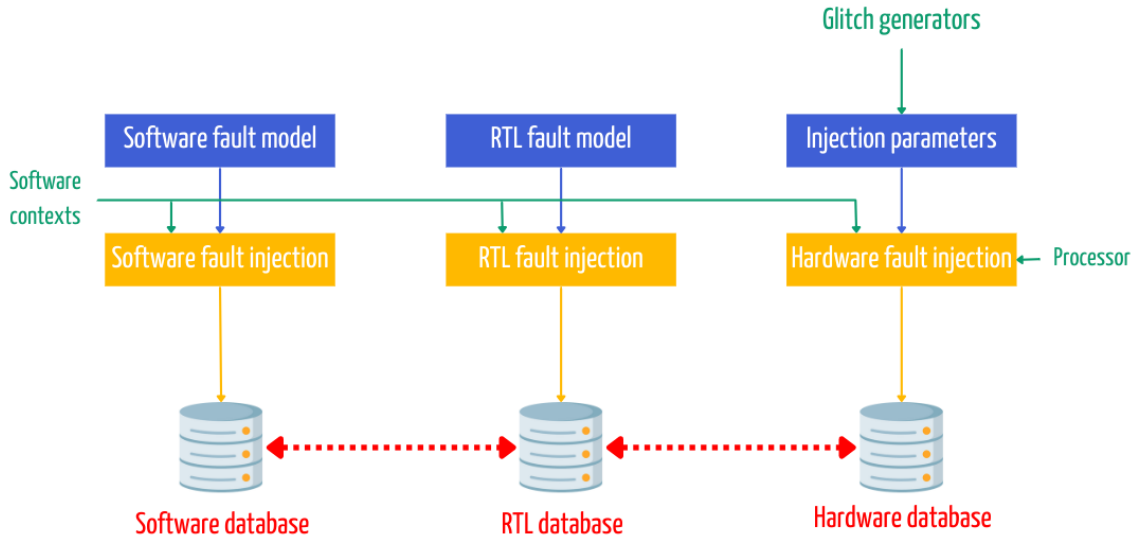


Figure 1.2: CLAM Methodology [1].

RTL problems are studied and used to assess software faults by targeting hardware targets (ARM processors). Software flaws, for example, whose effects are seldom detected, are removed, and new realistic ones are added. With this investigation, the proposed software and hardware countermeasures are double validated and their efficacy is improved [1].

## 1.5 Our contribution to the project

Protecting embedded systems against malicious usage is an urgent need because of the growing ubiquity of embedded systems in a variety of different fields of operation. When



designers use approaches that are based on hardware, they only have access to a limited number of injection places as well as injectable defects. As a consequence, the amount of observability and controllability that may be achieved is reduced.

As part of CLAM project, this work will entail simulating software fault models automatically using existing software fault models and additional models inferred from physical fault injections and developing an automated method to compare software fault simulation and physical injection, as well as software simulation and RTL (Fig. 1.2). This gives support to engineers as they conduct a series of more rigorous tests on the system under investigation in real time for two main reasons.

On one side, in order to develop the most effective countermeasures, a significant amount of adaptability is required. The use of software-based approaches is the only way to achieve this goal. In fact, as compared to methodologies based on hardware, they provide the advantage of incorporating any design error that may exist in the real hardware and software design while still keeping a low level of complexity and cost of implementation. In addition, the high-level fault characterisation may be expanded to include a wider variety of flaws if necessary.

On the other side, the flexibility of fault models is just as crucial as their level of realism. It is recommended to combine the adaptability of software simulation with the precision of hardware fault injection in order to get a deeper comprehension of the issue. This assists in having a more complete picture of the problematic flawed behaviors that have formed, the frequency with which they happened, and determining which ones have the potential to spark a collapse in the system.

## 1.6 Conclusion

This chapter provides an overview of the background of the project titled "Fault Injection and Simulation on Microprocessor Architectures," as well as the organism that offers it. This section concludes with a description of the issue being tackled, an overview of the project, and an analysis of my own participation in it. In the next chapter, the potential solutions that have been offered will be explored, focusing on the value that will be contributed by this project.

## Chapter 2

### Background and related work

## 2.1 Introduction

In order to facilitate a deeper comprehension of the information presented in this report, this chapter provides definitions of a few key terminology. After that, the associated study are explored while offering an explanation of the value offered by fault injection and simulation on microprocessor architectures.

## 2.2 Methodologies and Terms

### 2.2.1 Fault

Faults may be seen as the underlying cause of errors and other undesirable software behaviors. A fault is silent or ineffective if the error it generates does not spread and the program still exits properly, so at the end an expected output is observed. In contrast, if it influences the execution of the program and leads to a failure, which is defined as observable behavior different from that expected. A third case is when the program crashes causing errors or a reset. Ineffective fault implementations may be exploited, however, most successful fault attacks take use of system failures to leak sensitive data. In the event of a malfunction, it is possible that either the hardware or the software were at the source of the fault. However, most software bugs in production environment are temporary since the most enduring faults have been discovered in the early stages of the development process (design, review, and testing).

### 2.2.2 Fault model

A fault model is an abstract description of a fictitious malfunction. These models are utilized during fault analysis to identify possible implementation flaws and to predict the impacts of fault injection assaults, making them critical for higher-level attack strategy analysis. Because fault models are interpretations of observable behaviors, the same fault may be represented differently, and the degree of abstraction of the model can be selected based on the analysis level considered. In this project, Instruction Set Architecture fault models are used.

### 2.2.3 Fault injection

The term "fault injection" refers to a reliability verification approach whereby faults are intentionally introduced into a system through orchestrated experiments and the system's behavior is then observed.

### 2.2.4 Hardware-based fault injection

Hardware-implemented fault injection (HWIFI) is an approach of fault injection that uses simulated hardware failures to cause the system to fail and it's commonly used for

detecting IoT threats that depend on monitoring system-level properties and designing reliable electronic circuits. This kind of fault injection may be produced by a broad range of causes, including temperature, voltage, and clock glitches. However, HWIFI is seldom utilized since it is disruptive and hard to regulate, and because of its poor portability and observability.

### 2.2.5 Software-based fault injection

Software-based fault injection: Software fault injection (SFI) is an approach to fault simulation that alters the code. This approach involves using a specialized application to imitate the system's software and hardware problem conditions.

This method may be easily expanded to include other fault scenarios. Software injection aims to mimic the effects of physical and logical injection by inserting flaws at the functional level, altering the statements of program execution, adding, updating, or removing data, or directly affecting the contents of memory or registers. Unlike hardware fault injection, software fault injection may be performed with far more versatility and without the need for costly injection devices.

It can be broken into two subcategories:

- Static SFI where the change is made on source code level. It is also known as instruction-level faults where fault injection effect is replicated at the software program.
- Dynamic SFI or RTL level fault where the changes are made during execution time by setting off a trigger and then altering data in memory or registers.

As well as being cheaper and more portable than HWIFI, SFI also offers more flexibility and a broader range of implementation.

## 2.3 Related work

Fault injection testing has progressed from shorting out board connections to sophisticated techniques based on strong radiation that target particular hardware areas. When it became clear that hardware-based tests had several shortcomings, software based tests were presented as a workaround for these limitations basing on multi-dimensional abstraction layer between the software behavior and the conducted results on hardware level. The introduction of software analysis, made it possible to test a wider variety of solutions that were more portable across ISAs and microarchitectures.

Several researches have shown that fault injection attacks were able to affect the great majority of widely used standard chips with no exceptions made for fast processing. In addition, various architecture are vulnerable to such an attack (i.e. Intel, ARM...). Although the outcomes vary depending on the platform, an agreement can be seen that instruction substitution is the main source for this kind of assault.

The initial motivation for this work was to assess the effect of a fault injection on the running software by providing realistic and effective fault models while keeping the link between the three layers. In fact, at the ISA level, software models are often referred to the intended instruction directly, without taking into account any extra hardware limitations. For example, while discussing the skip model, the phenomenon in question is referred to as full instructions skip and considered as changing the instruction to a NOP [2]. When it comes to corruption, it is either the opcode or one of the operands (source or destination) that has been changed [2].

When taking the example of FiSim [3] which is a simulator for ISA-level faults, it still have a limited library of fault models. In fact, it only supports faults that can be modeled by modifying an already executed instruction or putting breakpoints on some instructions. In other words, it simulates instruction skip or bit flip models. Single bit faults are used to simulate attacks that change one bit of software each time. For instance, bit flip converts one bit from state to another, from 0 to 1 or the opposite. This mechanism is performed in a loop until covering all the software possible models. Furthermore, it does not provide hardware-specific findings which make it hard to prove the realism of the fault and its microarchitectural impacts. This approach is included in the software chip, thus not only can the analyst will not get to be certain of the realism of the fault models being simulated, but the performance of the smart card will also decline dramatically due to the time-consuming nature of simulating all possible models.

Trouchkine [4] contributed to the research effort by developing a granular set of test instructions capable of targeting individual components from embedded systems which reached the capacity of distinguishing between cache memory and management unit. Despite the fact that this technique suggested some plausible models, it was unable to provide an explanation for more intricate ones.

For instance, in [5], electromagnetic faults were performed on ARM Cortex-M3 boards while putting a significant amount of emphasis on introducing faults to one loaded instruction during the fetch stage. Nonetheless, the model remained inexplicable when the tainted data did not map to a single command.

A different approach was used in [6]. An injection module, placed on the target system, introduces fault during the code execution. Despite the realism of the fault propagation in the architecture level, this method corrupts registers rather than performing realistic faults. Thus, we can not be sure if the system is vulnerable to such attacks.

In this study, we take into account the mechanism by which instructions are fetched from the flash memory. In light of this methodology, we suggest new software models and provide estimates for the frequency with which these attacks is successful. In addition, an automated tool is provided for simulation, and results from the software models are compared to those from actual injections and RTL simulations to ensure they are realistic. This research will aid in the development of sophisticated and effective countermeasures, since the developer will be aware of which models may be performed on the target board and how probable it is to be achieved.

## 2.4 Conclusion

This chapter defines significant key words and terminologies. This was the starting - point for the conversation on the related work that was offered in the industry and how this research may bring value to the issues that are caused by fault injection attacks. Without further ado, the conception of the solution offered by this work is broken down in further depth in the next chapter.

## Chapter 3

# Software model simulation and analysis automation conception

## 3.1 Introduction

The first step in the development process is to explore the conception of the tool that will best meet the needs. As a result, the first section of this chapter begins by reviewing the motivations behind the tool studied and then go on to define its goals. After that, the software tool for automating model simulation and analysis will be presented, and its architecture will be broken down in depth. Before concluding this section, the steps involved in satisfying the requirement of the tool will be examined.

## 3.2 Need Recall

Due to the increasing prevalence of embedded systems in many fields of activity, safeguarding them against abuse is an absolute need. When adopting hardware-based methods, designers have only a limited set of injection spots as well as injectable faults, resulting in a restricted level of observability and controllability.

On one hand, to build optimum countermeasures, a considerable level of flexibility is a must. This is only achievable by incorporating software-based methods. In fact, compared to hardware-based techniques, they offer the benefit of integrating any design flaw that may exist in the actual hardware and software design while low maintaining complexity and implementation cost. Furthermore, the high-level of fault characterization can be extended to include additional types of defects [7].

On the other, realism in fault models is as important as adaptability. To better understand the fault, the versatility of software simulation should be paired with the accuracy of hardware fault injection. This helps in having a fuller picture of the problematic faulty behaviors that have emerged, the frequency with which they occurred and determine which ones potentially trigger a system breakdown. To achieve this goal, an automated comparative solution should be developed to assist engineers in running a more rigorous set of tests on the system under investigation in real time.

## 3.3 Software model simulation and analysis automation goals

The fulfillment of our goal may be divided into two separate steps, each of which is outlined in the figure 3.1.

1. **Realistic fault model simulation** : To accomplish this purpose, realistic fault models should be simulated. Other team members investigate the problem in detail at the micro-architectural level after getting knowledge from the visible failures caused by the introduction of physical flaws. This helps in determining to what extent the problem lies inside the system itself, hence proving the realism of the suggested fault models and providing new ones. Once decided, software models should be developed while simulating the microprocessor's behavior. In reality,



the target assembly code are executed at a software level , without the need of any special-purpose hardware. So, while making modifications to our objective, all conceivable scenarios should be investigated and considered in the same manner as a microprocessor would; This includes crash instances that are handled differently; instead of having errors in the Python code, the user should be notified of the model causing the crash. At the end of this step, a CSV file including all of the software model simulation results is generated.

2. **Analysis automation** : The CSV file that contains the results of the software fault models generated in the prior phase is compared to the CSV file that contains either the outcomes of the physical fault injection or to the RTL simulation CSV file outcomes. A final report is generated. For each fault injection outcome, the user is informed of how many software fault models match and give the same registers values. All conceivable software models are numbered and described in depth. And because the occurrence percent of each software model is an important factor, the user is also informed of the models that have a greater likelihood of succeeding. Last but not least, the number of crashes seen for each delay is shown to the user. This information is useful in the process of building optimum countermeasures.

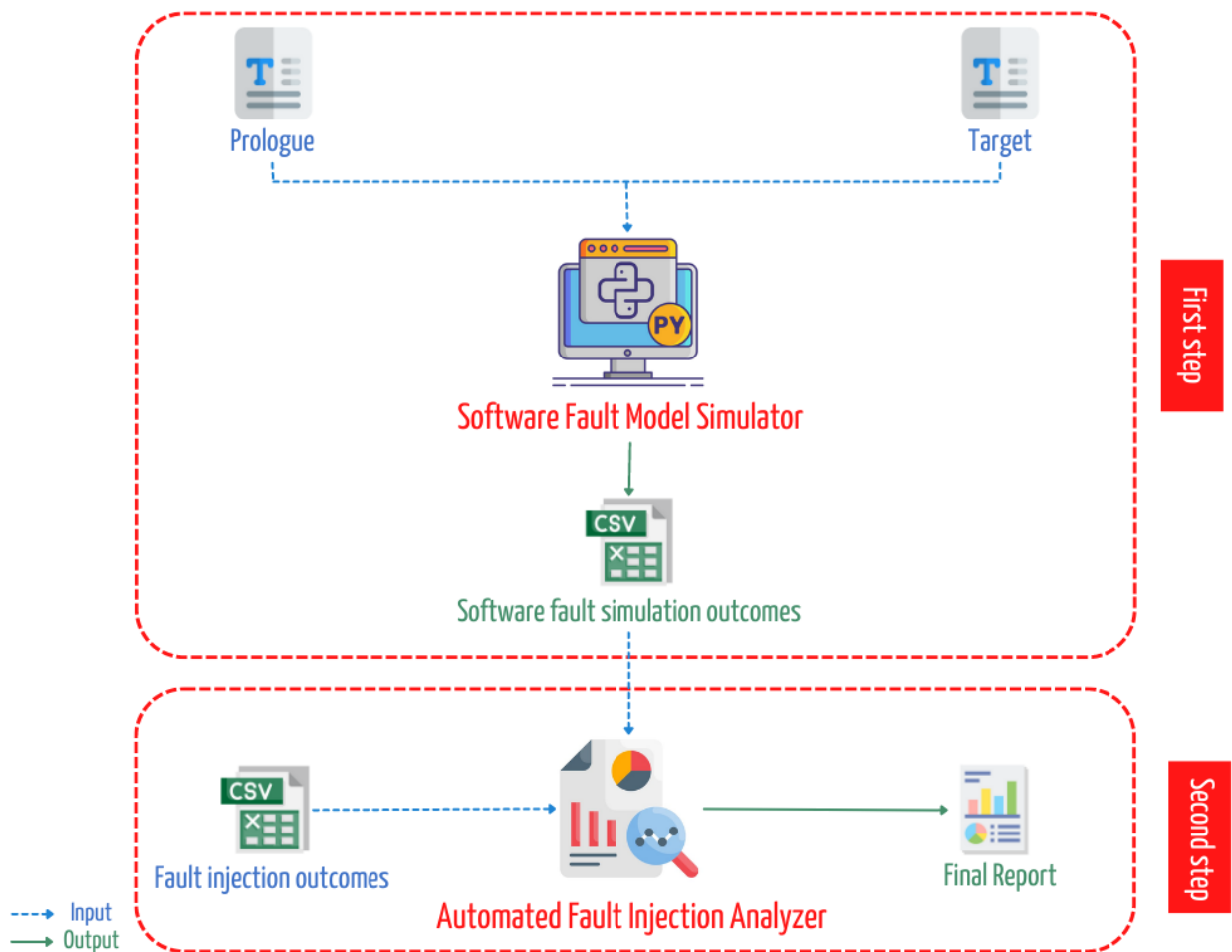


Figure 3.1: Software model simulation and analysis automation goal.

### 3.4 Software model simulation and analysis automation presentation

A solution is provided in the form of a Linux-compatible Python script that interacts with many modules in order to accomplish its objective in the manner shown in the figure 3.2.

During the first stage of the procedure, the user is prompted to provide the target code file as well as the register initialization file. Every time a software fault model module is executed, the software fault model simulator module re-initializes registers with the provides initialization file, performs specific changes to the target code and connects with the microprocessor emulator to simulate the software fault model.

Finally the outcomes of the models are saved to a csv file. Once all of the fault models are executed, the user should provide the CSV file containing the data of the fault injection that they want to investigate. A line by line comparison is performed by the automated fault injection analyser module. Each time a fault injection outcome matches a software fault model result, the details are appended to a final report that will be provided to the user.

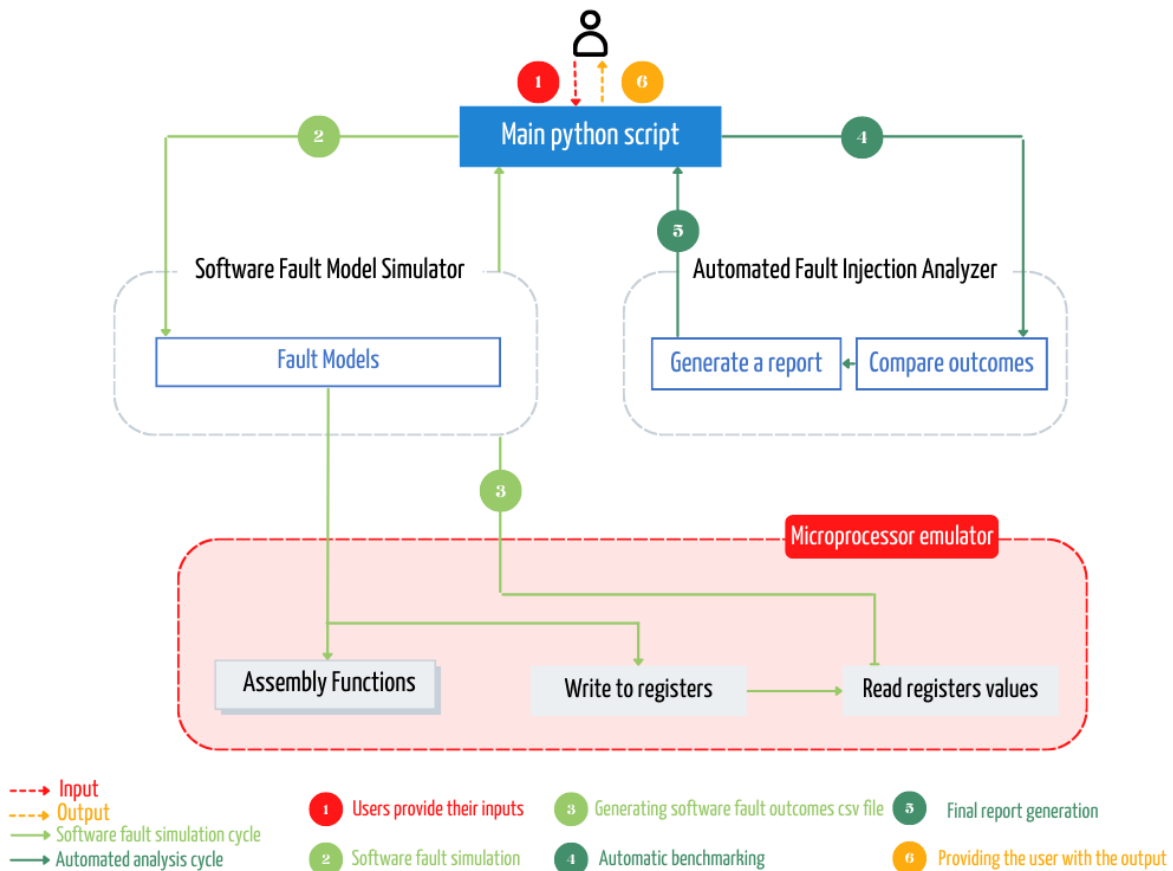


Figure 3.2: Software model simulator and analysis automator.

### 3.5 Software model simulator and analysis automator's functional architecture

In order to put the suggested solution into action, the modules presented in the figure 3.3 are required:

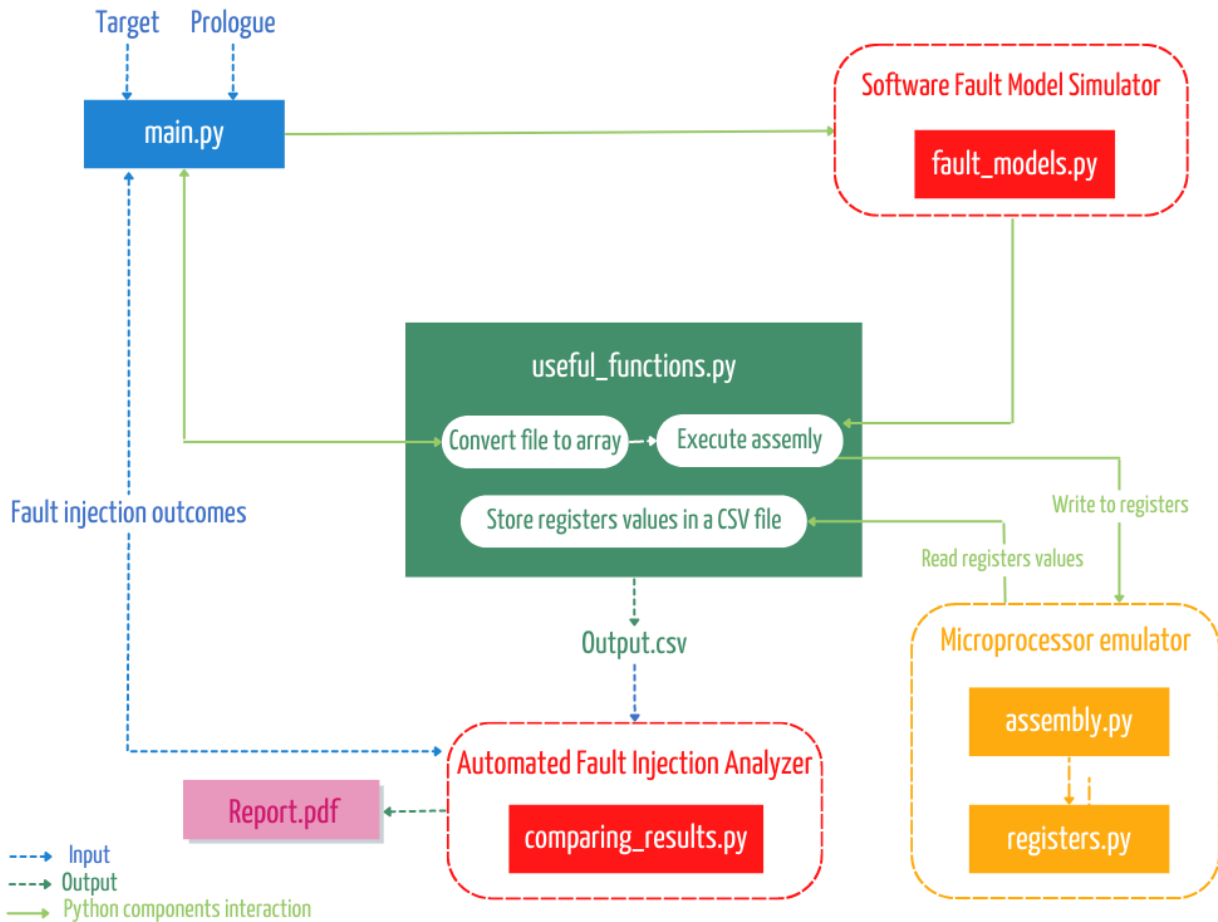


Figure 3.3: Software model simulator and analysis automator's functional architecture.

1. **main.py** : This is where our application begins. This script is responsible for importing all of the other modules and connecting with them, and it serves as the principal execution environment with which the user interacts.
2. **Software fault model simulator module** : This module includes the fault model specifications in their entirety. It incorporates all of the functions that were designed to alter the target code file in such a manner that it simulates the fault behaviour desired.
3. **Automated fault injection analyzer module** : This module is in charge of the comparison analysis. When the user supplies the fault injection results file (physical injection or RTL simulation), it is evaluated line by line and compared to the previously prepared software simulation outcomes file. The results are documented in

a final report. Fault injections are matched with the corresponding software models. A thorough explanation of each model, as well as the percentage of time these outcomes were seen throughout the fault injection campaign, are supplied. In the end, it tallies up the number of crashes found throughout the course of the whole campaign for each delay separately.

4. **Microprocessor emulator module** : In order to simulate the software models correctly with no need to physical connection to a board, the microcontroller should be emulated and the CPU behaviour should be expected for each case. The first axiomatic fact that should be handled, is the registers. This refers to a location in memory that may be written to or read from. In our case, registers are represented as global variables having both module-level access, as well as global access for both reading and writing. It is recommended to define them as global variables in their own module. Second, the target code is a set of assembly instructions that a standard Python interpreter can not translate into a binary. This means, personalised functions should be created to mimic the assembly language's functionality while keeping the syntax as simple as feasible.
5. **useful\_functions.py**: This module provides some important functions for appropriately simulating the fault. For example, it converts the user-provided file into an array so that the process may be carried out more easily. In addition, there is a specific syntax that must be followed while working with global variables. This goes against the making assembly syntax more accessible to the public. In order to remedy this situation, a function that modifies the syntax of the assembly function has been developed. For example, the assembly instruction "ADD(R0, R1, 50)" is transformed to "registers.R0 = add.w(registers.R0, registers.R1, 50)". Aside from that, the amount of time taken to run a fault model is traced and monitored. If it takes longer than a certain amount of time, it is considered as a crash in order to prevent endless loops from occurring. Last but not least, it is the responsibility of this module to write to and read from registers in order to store the results of each software failure model simulation.

### 3.6 Software model simulator and analysis automator's functional process

The software fault model simulator and analysis automator follows the approach shown in the figure 3.4 to accomplish its purpose. To properly run the fault models, our emulator should be placed under the same circumstances as the target hardware. In particular, the identical values are stored in each register before performing the attack. The first step is to use the prologue file as input and convert it into an array. This array of assembly instructions is executed, and the values of the registers are set to their initial values; a new column in the output CSV file is appended with the heading "Initial," which stands for initialization registers values. The target code file should be given during the following step. In a similar manner, it is serialized as an array, run, and the output is concatenated to the final CSV file with the header "Golden" denoting the golden run. Each simulation

begins with all registers reset to their initial values and then the fault model execution. The output file is updated with the results of each simulation. The user may then choose to evaluate the results of either the physical injection or the RTL simulation and provide the CSV file holding the outcome results. This file is compared to the results of the fault models and the final report is produced.

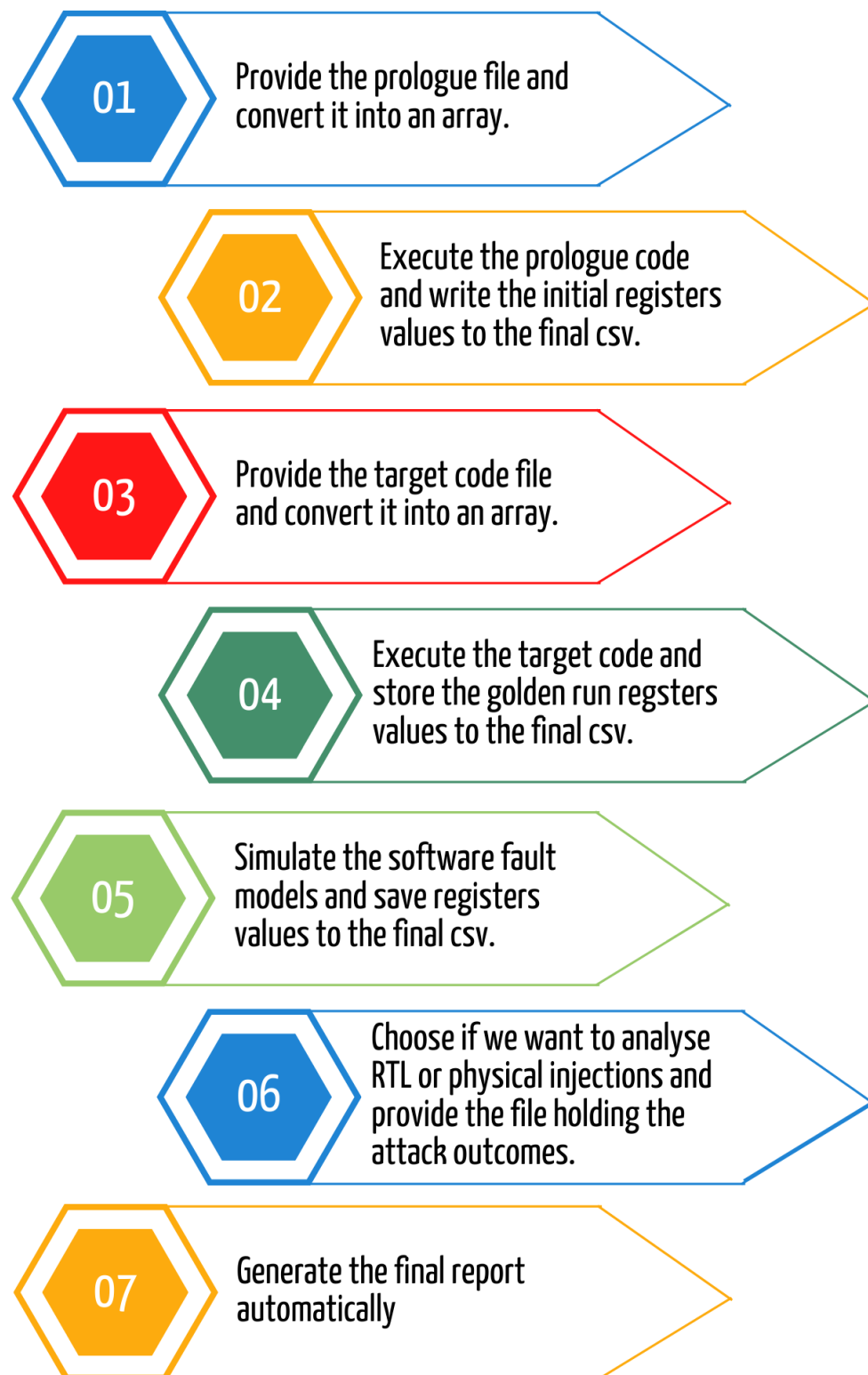


Figure 3.4: Software model simulator and analysis automator's functional process.

## 3.7 Conclusion

In this chapter, the notion of the tool was broken down in further detail. Prior to going on to the process of articulating its goals, the motivations behind the software application that automates model simulation and analysis were examined. After that comes the presentation of the software tool for automated model simulation and analysis, followed by an in-depth breakdown of the product's design and the procedures that must be taken in order to fulfill the criteria of the tool. The next chapter serves as an introduction to the tool's development process by setting up the suitable environments.

## Chapter 4

### Environment setup



## 4.1 Introduction

This chapter will go through the requirements for the software fault model simulator and analysis automator before moving on to the next section, which will cover the development process. The first part of this section will focus on defining all of the used modules and technologies. During the second stage, the established assembly syntax will be explained. At long last, the target board that was emulated during the course of this study will be discussed, along with the characteristics that define it.

## 4.2 Technologies and modules used

### Python

Python<sup>1</sup> is a well-known interpreted programming language due to its large library, strong numerical approach, and standing as a data processing icon. Python has great packaging, composability, and embeddability features, and it can readily accommodate a broad variety of complex structures which makes it adaptable to many situations. Python's enormous number of libraries make it a simple yet effective language for scripting and automation.

- High capacity for managing data
- Easy integration with other programming requirements, such as CPython
- Python produces comprehensive functionality with a short amount of code



Figure 4.1: Python Logo.

### Pandas

Pandas<sup>2</sup> is an open-source Python toolkit that provides high-performance data structures for statistical processing and analysis. Pandas have various appealing qualities, including:

- Highly performant DataFrame objects, with support for both built-in and user-defined indexes.
- Alignment of data and missing data processing are handled together seamlessly.



Figure 4.2: Pandas Logo.

---

<sup>1</sup><https://www.python.org/>.

<sup>2</sup><https://pandas.pydata.org/>.

- Reshape and set the dataset.
- Large datasets may be indexed, subset, and sliced depending on labels.
- High-speed data-joining and combining capabilities.

## Numpy

NumPy<sup>3</sup> is the backbone of Python's scientific computing ecosystem. It is a Python library that offers discrete Fourier transform, elementary linear algebra, elementary statistical operations and stochastic simulation, and more as well as derived objects like masked arrays and matrices. NumPy arrays, in contrast to lists, are kept in a single continuous region of memory, making them easy for programs to access and edit. In addition, Numpy provides a number of tools for incorporating low-level code, such C and C++, that may improve efficiency when working with complicated calculations.



Figure 4.3: Numpy Logo.

## Jupyter

Jupyter<sup>4</sup> is an electronic notebook project that enables users to recombine different elements, including text, graphs, mathematical computations, and so on. Modifications are made possible in an interactive way by using a web browser. With the forty programming languages it supports, it offers a versatile user interface. For our purposes, It is used for fault injection observations and manipulations.



Figure 4.4: Jupyter Logo.

## FPDF

FPDF<sup>5</sup> python library for creating PDF documents. In comparison to other PDF libraries, PyFPDF is straightforward, compact, and general-purpose library, equipped with sophisticated capabilities that are simple to learn, expand, and maintain such as images integration, a visual designer and a simple html2pdf template.



Figure 4.5: FPDF Logo.

<sup>3</sup><https://numpy.org/>.

<sup>4</sup><https://jupyter.org/>.

<sup>5</sup><https://pypi.org/project/fpdf/>.

## Capstone

Capstone<sup>6</sup> is an easy-to-use disassembly framework that works on a wide variety of platforms and architectures. It Reveals specifics details regarding the purpose of the disassembled instruction, such a rundown of the registers that were read and written implicitly. It provides a simple, lightweight, and user-friendly API that is architecture-agnostic.



Figure 4.6: Capstone Logo.

## Keystone Engine

Keystone Engine<sup>7</sup> is a Python assembler framework that supports a range of architectures and works across several platforms, while also providing an intuitive API equipped with a variety of functions.



Figure 4.7: Keystone Engine Logo.

## 4.3 Assembly Functions

When developing our assembly instructions, Micropython served as a benchmark, frequently referenced while modeling this tool. Before digging into the produced assembly, Micropython is explained in the following.

### 4.3.1 Micropython

Micropython<sup>8</sup> is a lightweight variant of Python 3.4 that can operate on low-powered devices like microcontrollers, which typically have just 16K of RAM available during execution. In order to make the standard library work with embedded microcontrollers, many of the library's functions and classes have been removed. MicroPython was originally intended to operate on the pyboard<sup>9</sup> microcontroller, but it is now compatible with a wide variety of ARM-based embedded systems, including Arduino, allowing for the rapid creation of Python-based applications for things like automated control and robotics.

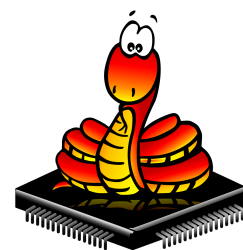


Figure 4.8: Micropython Logo.

<sup>6</sup><https://www.capstone-engine.org/>.

<sup>7</sup><https://www.keystone-engine.org/>.

<sup>8</sup><https://micropython.org/>.

<sup>9</sup>Small electrical circuit board that runs MicroPython

### 4.3.2 Assembly syntax

Micropython seems to be a powerful tool for our goal, since it combines our python need with the ability to operate on microcontrollers. Although this is a powerful solution that eliminates the need for further work: microcontroller emulation is not anticipated and assembly syntax development is not necessary; therefore the model was abandoned due to its numerous flaws. To begin, as stated in the description of micropython, it is designed to be used on pybord, which is a microcontroller that supports micropython. Unfortunately, not all targets currently support it. As an example, python development environments only support a restricted set of targets. Thus, the Python code must be created in a text editor and executed from the shell, which adds complexity to the testing process, especially with non-supported Python modules. Worse, the given solution is not scalable since it only works on a specified list of targets and adds additional overhead to the system due to the physical connection with the board. Furthermore, since it is still a small community with minimal documentation, along with being unable to read and write to registers other than R0 even though the sample code was properly executed on the cortex-M4 board.

Nonetheless, the existing defined assembly syntax was used as inspiration for our functions presented in the figure 4.9.

Move	Load & Store	Arithmetic	Logical	Shift & Rotate	Comparison	Branch
MOV(Rd, n)	LDR(Rt,Rn,imm7)	ADD(Rd,Rn,n)	AND_(Rd, Rn)	LSL(Rd, Rn, imm)	CMP(Rn, n)	B(label)
MOVS(Rd,n)	LDRB(Rt,Rn,imm5)	ADDS(Rd,Rn,n)	ANDS(Rd, Rn)	LSLS(Rd, Rn, imm)	CMN(Rn, Rm)	BNE(label)
MOVW(Rd,imm16)	LDRH(Rt,Rn,imm6)	SUB(Rd,Rn,n)	ORR(Rd, Rn)	LSR(Rd, Rn, imm)	TST(Rn, Rm)	BGE(label)
MOVT(Rd,imm16)	STR_(Rt,Rn,imm7)	SUBS(Rd,Rn,n)	ORRS(Rd, Rn)	LSRS(Rd,Rn,imm)		BGT(label)
MOVWT(Rd,im32)	STRB(Rt,Rn,imm5)	NEG(Rd,Rn)	ERO(Rd, Rn)	ASR(Rd, Rn, imm)		BLE(label)
	STRH(Rt,Rn,imm6)	UDIV(Rd,Rn,Rm)	BIC(S)(Rd, Rn)	ROL(Rd, Rn, imm)		BLT(label)
		SDIV(Rd,Rn,Rm)	EROS(Rd, Rn)	ROR(Rd, Rn, imm)		
		MUL(Rd,Rn)	MVN(S)(Rd, Rn)	RBIT(Rd, Rn)		

Figure 4.9: Micropython syntax for Thumb-2.

The syntax follows this pattern: <sup>10</sup>

**Opcode**(**Destination Operand** , **Source Operand(s)** )

For example:

- **MOV**(**R0** , **R1** ) stands for **R0** = **R1**
- **SUB**(**R1** , **R3** , **5**) stands for : **R1** = **R3** - **5**

<sup>10</sup>[https://docs.micropython.org/en/latest/reference/asm\\_thumb2\\_index.html](https://docs.micropython.org/en/latest/reference/asm_thumb2_index.html)

- `ADD(R2, R5, R6, LSL, 2)` stands for :  $R2 = R5 + (R6 \ll 2)$  where the  $\{\ll 2\}$  refers to shift left by two bits.

## 4.4 Target board

Although some fault models may be easily described at the instruction set architecture (ISA) level, others need knowledge of the binary encoding of the instructions in order to provide an explanation for the observed behavior. This approach was used in [8] to gain a deeper understanding of certain previously unexplained fault models and suggest potential new ones.

As a result, before digging into the intricacies of the fault models contained in our automated solution, it is critical to have a basic grasp of a few essential concepts as well as how the instructions are stored inside the memory.

The target board used is a 32-bit microcontroller. It incorporates an ARM Cortex-M3 processor and supports the THUMB2 instruction set (ISA)<sup>11</sup>. As a considerable upgrade to the original Thumb ISA, Thumb-2 has several new features. This is achieved by the introduction of new 32-bit instructions combined with the earlier 16-bit Thumb instructions<sup>12</sup> [10] [11].

Cortex-M3 has a three-stage pipeline that allows it to load, read, and then execute a set of instructions. This is what is referred to as the instruction cycle or the fetch, decode, and execute cycle as demonstrated in the figure 4.10.

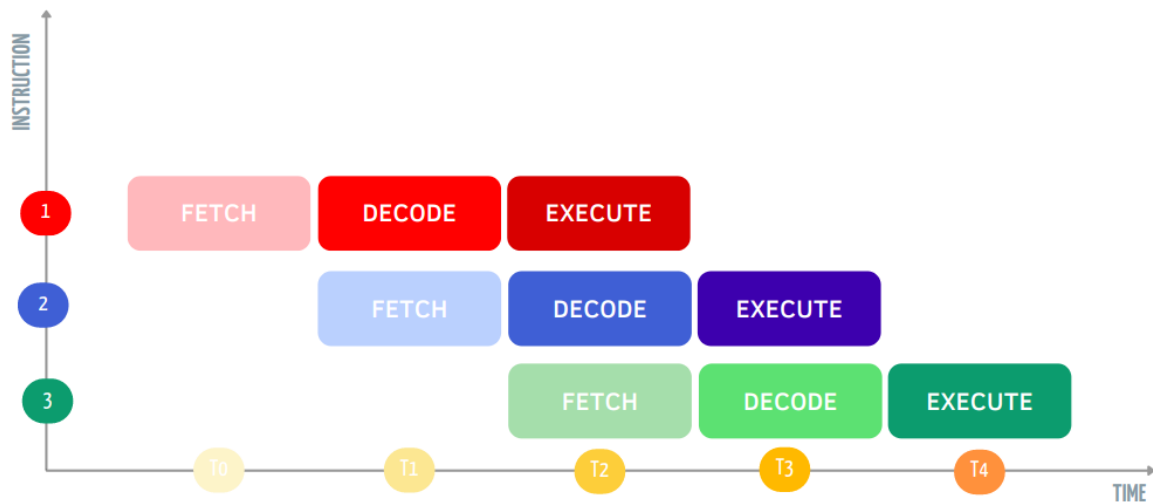


Figure 4.10: Instruction cycle.

<sup>11</sup>The tool may be used to evaluate numerous board outcomes. It has been tested on 32-bit MCUs: cortex-m0, cortex-m3, and cortex-m4.

<sup>12</sup>THUMB instruction set architecture was added to ARM as a second instruction decoder to deliver higher code density. In fact, despite being just 16 bits long, it is compatible with 32-bit ARM instructions. [9]

1. **Fetch** : In the fetch procedure, the central processing unit (CPU) transfers data from the Program Counter(PC) to the memory address register (MAR) through the data bus.

During the fetch stage, the CPU delivers the content of the PC to the MAR through the data bus. In return, the MAR transmits the 32-bit word that is located at the given address (equivalent to PC). In other words, regardless of the length of the instruction, the fetch size is always limited to 32 bits per cycle [12] as demonstrated in the figure 4.11.

For example, two 16-bit instructions may be retrieved or one 32-bit instruction. This is the scenario where memory contains an aligned code, in which each 32-bit data corresponds to the same 32-bit instruction or to two 16-bit instructions [8].

However, the alternative instance of misaligned code might exist, where the 32-bit data fetched does not match to the same instruction, whether it is a 16-bit instruction coupled with the most significant 16 bits of a 32-bit instruction, or the opposite scenario where the least significant 16 bits of a 32-bit instruction are combined with a 16-bit instruction, or even combining the bottom half of a 32-bit instruction with the top half of another 32-bit instruction [8], as illustrated in the figure 4.11.

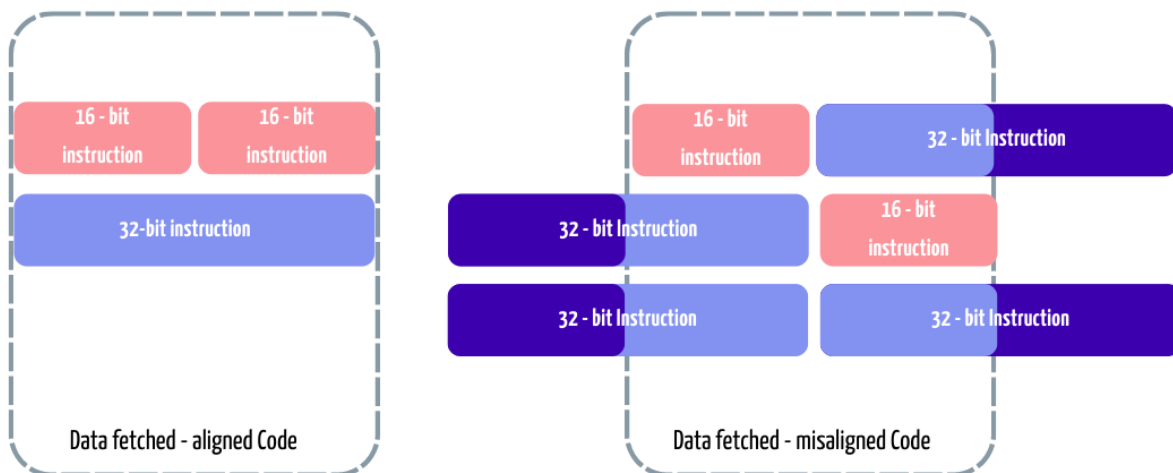


Figure 4.11: Instruction cycle - Fetch Stage.

2. **Decode** : During this stage, the CPU must identify which instruction should be executed based on its encoding, as well as the number of operands to be fetched for this purpose<sup>13</sup>. The opcode that was retrieved from memory is then decoded in preparation for the following phases and relocated to the appropriate registers.
3. **Execute** : After the instruction has been decoded, it is sent to the appropriate CPU functional units, which then carry out the necessary operations, such as retrieving data from registers, processing it in the ALU, and then returning the result to a register.

<sup>13</sup>Some instructions, such as the addition instruction, have one destination operand and two source operands. Others, such as move instructions, have one destination operand and one source operand.

4. **Repeat** : This fetch decode execute cycle is continued until no further instructions are found.

## 4.5 Conclusion

After ensuring that everything is properly installed and that our assembly syntax is developed, as was demonstrated in this chapter, it is time to begin the process of developing the tool itself. The following chapter will go into further detail about this topic.

## Chapter 5

# Software model simulation and analysis automation development



## 5.1 Introduction

This chapter is able to be broken up into two significant parts. First, the developed software models will each be thoroughly described. After that, an introduction to the automation module will be provided with an overview of its features.

## 5.2 Software fault models

### 5.2.1 Instruction Skip

This is one of the most well-known models. It entails skipping one or more instructions from our target program.

Taking into consideration the scenario shown in the figure 5.1, one 32-bit instruction, and two 16-bit instructions were skipped respectively in the first and second illustration. Both of the target programs that were demonstrated provide an aligned code. To accomplish this model, the CPU skipped a 32-bit word corresponding to the same instruction in the first example and two consecutive 16-bit instructions in the second.

It's important to keep in mind that the number of possible skipped instructions is related to the cache size. For instance, with a 128-bit board [13], it is possible to skip up to four 32-bit instructions, eight 16-bit instructions, or a combination of both (three 32-bit instructions and two 16-bit instructions for example).

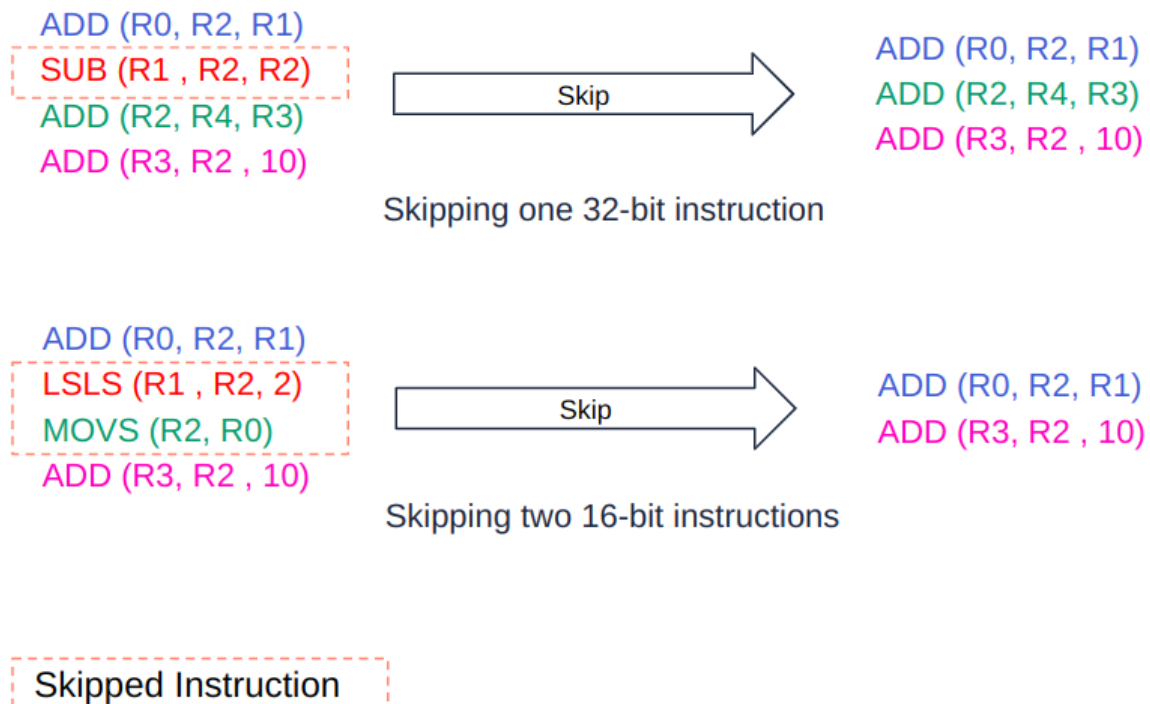


Figure 5.1: Skip fault model.

## 5.2.2 Instruction Skip and instruction Repeat

This model involves skipping one or more instructions and then repeating one or more previously executed instructions.

Three examples of this model's applicability are shown in the figure 5.2. In the first example, a word containing two 16-bit instructions was bypassed and substituted with a single 32-bit instruction. The second case illustrates the reverse scenario, in which one 32-bit instruction is skipped and a word made up of two 16-bit instructions is repeated. Furthermore, the final example demonstrates that the instructions skipped and the repeated ones do not need to be consecutive. In other words, all that's required is to repeat instructions that have previously been executed.



Figure 5.2: Skip and repeat fault model.

Once again, the number of instructions that may be skipped is limited by the available memory, while the number of repeated instructions is proportional to the number of instructions that were skipped.

In fact, the lowest integral value greater than the number of skipped instructions divided by two is the value that determines the minimum number of repeated instructions. For instance, if three instructions are omitted (a 32-bit instruction and two 16-bit instructions, which together add up to 64 bits), it is required to repeat at least two instructions (two 32-bit instructions totaling 64 bits). The maximum number of repeated instructions is two times the number of skipped instructions. For example, if two 32-bit instructions (bringing the total to 64 bits) are omitted, up to four instructions could be repeated (four 16-bit instructions for a 64 bits total).

### 5.2.3 Double instruction corruption

This model might be used in two separate scenarios, with differing outcomes. The fundamental idea is same in both cases. This concept is described when instruction encoding is examined. In this fault model a misaligned target code<sup>1</sup> is used.

For this demonstration, a sample target program having a 16-bit instruction followed by 32-bit instructions is used. The first line includes the 16-bit instruction, followed by the most significant 16 bits of a 32-bit instruction. The following lines include a 32-bit instruction's least significant 16 bits joined with the next 32-bit instruction's most significant 16 bits.

In this model, 32 bits (that are not part of the same instruction) are skipped and then the previous 32 bits<sup>2</sup> are repeated. There are two possible outcomes, depending on whether the first 16 bits repeated represent the initial 16-bit instruction or if they belong to a 32-bit instruction.

#### Double instruction corruption : Repeating bits that belong to 32-bit instruction

In this scenario, the result indicated in the figure 5.3 is acquired by skipping 32-bit line and repeating the previous 32-bit line.

As a consequence, when the instructions are reassembled, the following will occur:

- The first 16-bit instruction will be successfully executed (the blue one).
- The second instruction's most significant 16 bits will be correctly combined with the instruction's least significant bits (red one). As a result, the second instruction will be executed appropriately.
- The most significant 16 bits of the third instruction (`\xbe\xa0`) will be combined with the repeated 16 bits of the second instruction (`\x01\x03`). As a result, since the most significant bits of a 32-bit instruction refer to the opcode and the first source operand, while the least significant ones refer to the destination operand and

<sup>1</sup>Words in the memory don't map to individual instructions.

<sup>2</sup>In this example, 32 bits are skipped and another 32 bits are repeated for clarity, but the model may be expanded, by skipping and repeating more bits, using the same concept.

second source operand, the third instruction will be corrupted. Its destination and second source operands will be substituted by those of the second instruction.

- The repeated most significant 16 bits of the third instruction (`\xeb\x0a`) will be combined with the least significant 16 bits of the fourth instruction (`\x03\x0a`). Therefore, the corrupted third instruction, whose destination operand and second source operand have been substituted with those of the fourth instruction (pink one), will be executed instead.

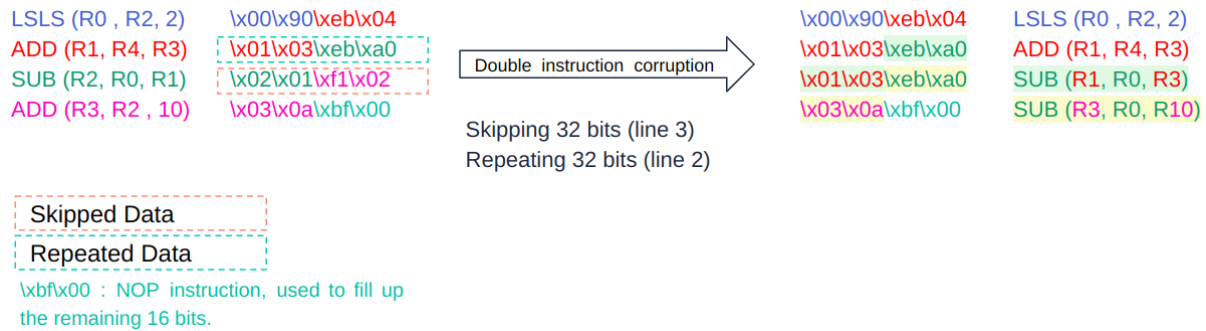


Figure 5.3: Double instruction corruption fault model - case 1.

### Double instruction corruption : Repeating bits that belong to 16-bit instruction

This scenario is similar to the previous one. It is obtained by skipping the second line and then repeating the first one as indicated in the figure 5.4.

As a consequence, when the instructions are reassembled the following will occur:

- The first 16-bit instruction will be successfully executed (the blue one).
- The second instruction's most significant 16 bits (`\xeb\x04`) will be combined with (`\x00\x90`) which is the first 16-bit instruction. Oppositely to the first situation, the corrupted instruction will not retrieve the destination operand and second source operand, instead they will be deducted from the repeated 16-bit instruction. Note in this example, when assembling the obtained 32-bit instruction, a logical shift was performed.
- The repeated most significant 16 bits of the second instruction (`\xeb\x04`) will be combined with the least significant 16 bits of the third instruction (`\x02\x01`). Therefore, the corrupted second instruction, whose destination operand and second source operand have been substituted with those of the third instruction (green one), will be executed instead.

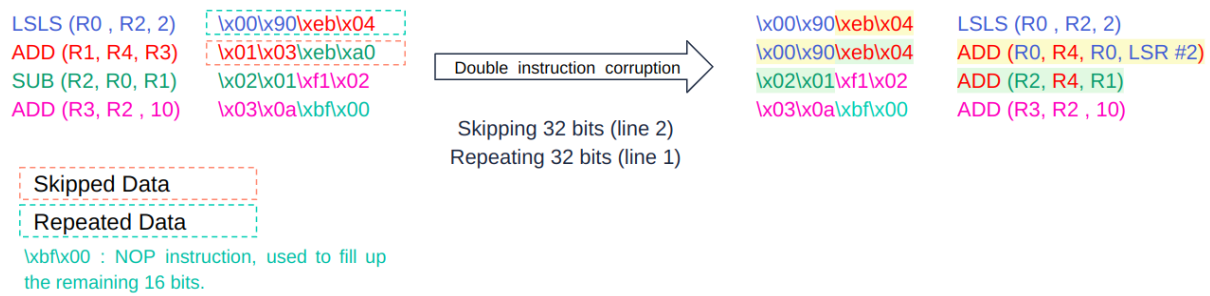


Figure 5.4: Double instruction corruption fault model - case 2.

### 5.2.4 Instruction skip, instruction repeat and double instruction corruption

This model is similar to the previous one, the double instruction corruption model. The difference is that in this case multiple lines are skipped and repeated instead of one. As a result, the observed behaviour will be different.

#### The first 16 bits repeated belong to a 32-bit instruction

In this scenario, the result indicated in the figure 5.5 is acquired by skipping the fourth and fifth lines and repeating the second and the third<sup>3</sup>.

As a consequence, when the instructions are reassembled, the following will occur:

- The first three instructions will be reassembled correctly (the red, blue and green ones)
- The most significant 16 bits of the fourth instruction (`\xf1\x02`) will be combined with the first line repeated, and so, the least significant 16 bits of the second instruction (`\x01\x03`). As a result, the fourth instruction will be corrupted. Its destination and second source operands will be substituted by those of the second instruction.
- Because the second and third lines were duplicated, the third instruction (green one) will be reassembled and re-executed. As a consequence, it will be executed a second time.
- The repeated most significant 16 bits of the fourth instruction (`\xf1\x02`)<sup>4</sup> will be combined with the least significant 16 bits of the sixth instruction (`\x05\x03`). Therefore, the corrupted fourth instruction, whose destination operand and second source operand have been substituted with those of the sixth instruction (purple one), will be executed instead. Note that the fifth instruction (gray one) was entirely skipped since its encoding is among the lines that were not processed.

<sup>3</sup>The example of skipping and repeating two lines was explored for clarity, but the model may be expanded using the same idea.

<sup>4</sup>Caused by the third line being repeated.

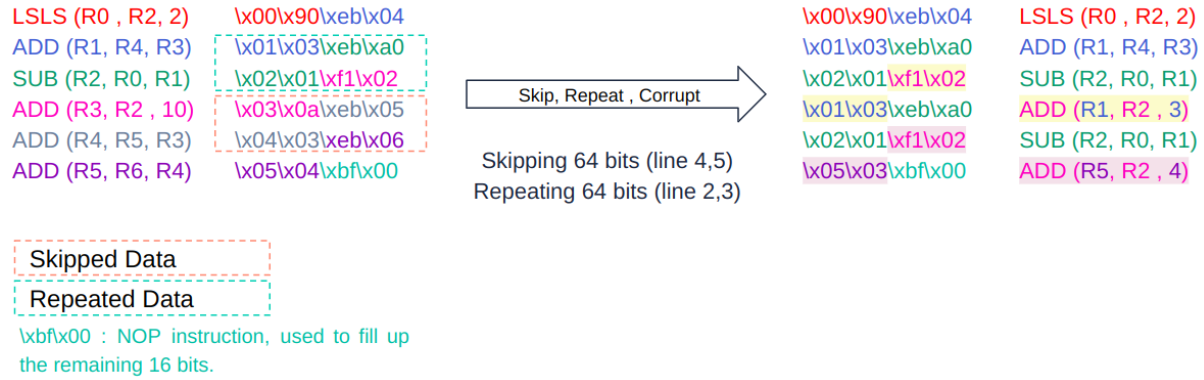


Figure 5.5: Instruction skip, instruction repeat and new instruction execution - case 1.

### The first 16 bits repeated represent a 16-bit instruction

This scenario is similar to the previous one. It is obtained by skipping the third and fourth lines, then repeating the first and second; as indicated in the figure 5.6.

As a consequence, when the instructions are reassembled the following will occur:

- The first two instruction (blue and red ones) will be successfully executed.
- The third instruction's most significant 16 bits (\xeb\xa0) will be combined with (\x00\x90) which is the first 16-bit instruction. Oppositely to the first situation, the corrupted instruction will not retrieve the destination operand and second source operand, instead they will be deducted from the repeated 16-bit instruction.
- Because the first and second lines were duplicated, the second instruction (red one) will be reassembled and re-executed. As a consequence, it will be repeated.
- The most significant 16 bits of the third instruction (\xeb\xa0) will be combined with the least significant 16 bits of the fifth instruction (\x04\x03). As a result, the third instruction will be corrupted. Its destination and second source operands will be substituted by those of the fifth instruction. Note that the fourth instruction (purple one) was entirely skipped since its encoding is among the lines that were not processed.

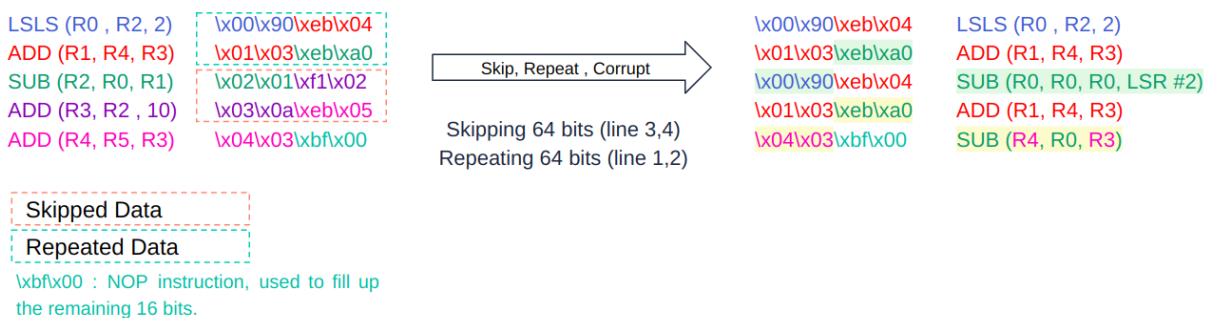


Figure 5.6: Instruction skip, instruction repeat and new instruction execution - case 2.

### 5.2.5 Instruction skip and instruction corruption

Because this fault model is also demonstrated on a misaligned code, the same sample target as in the prior models could be used. Within this context, one or more lines are skipped. For example, in the scenario demonstrated in the figure 5.7, 64 bits are bypassed and hence two lines.

When re-combining instructions, the most significant bits of the second instruction (`\xeb\x04`) will be combined with the least significant bits of the fourth instruction (`\x03\x0a`). Thus the second instruction (blue one) will be corrupted. Its destination and second source operands will be substituted by those of the fourth instruction. Furthermore, the instructions that come between the merged instructions (the second and fourth) will be skipped. In this scenario, just the third instruction will be ignored (green one).

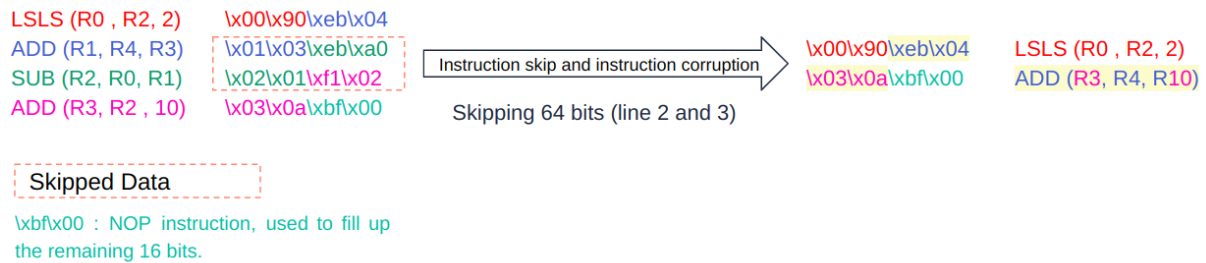


Figure 5.7: Instruction skip and instruction corruption fault model.

### 5.2.6 Instruction skip and new instruction execution

This model is comparable to the instruction skip and corruption model. Likewise, one or more lines are skipped. However, in this situation, the first skipped line is a combination of a 16-bit instruction and the top half of a 32-bit instruction (Fig 5.8). Therefore, the 16-bit instruction (blue one) will be skipped, and the least significant 16 bits remaining from the previous skipped 32-bit instruction (`\x01\x03`) will be considered as a new 16-bit instruction (`LSLS(R3, R0, 4)`).

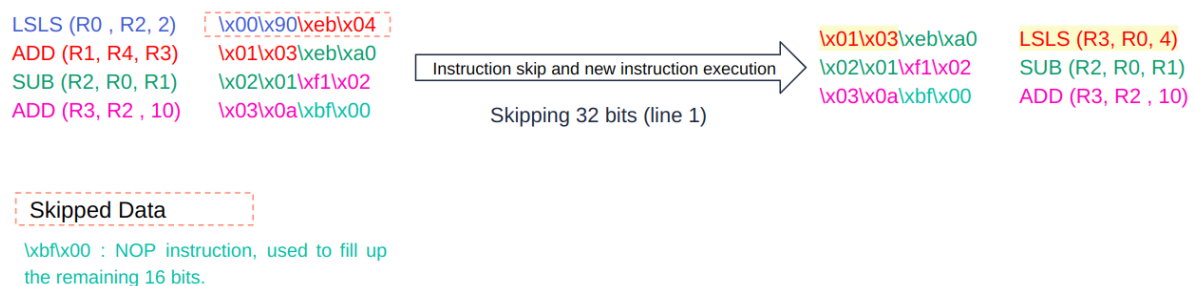


Figure 5.8: Instruction skip and new instruction execution fault model.



## 5.2.7 Instruction corruption with zeros

This model is quite similar to "instruction skip and instruction corruption" and "Instruction skip and new instruction execution," except that after skipping the specified number of lines, a sequence of zeros will be inserted.

To demonstrate this paradigm, two examples are included. The typical step is to skip a 32-bit line composed of the 16 least significant bits of a 32-bit instruction and another 16 bits belonging to the next instruction. Referring to the examples demonstrated in the figures 5.9 and 5.10 the second line is skipped. The most significant 16 bits remaining from the corrupted instruction (the second one in this example) are filled with zeros. As a result, its destination and second source operands will be modified by R0.

The only variation between the two cases relates to the least significant 16 bits of the skipped line. For instance, in the first case, they belong to a 32-bit instruction (the third instruction in this case). Its remaining least significant 16-bits (`\x02\x01`) are considered as a new 16-bit instruction (`LSLS(R1, R0, 8)`) as demonstrated in the figure 5.9.

In contrast, in the second case, the least significant 16 bits skipped refer to a 16-bit instruction which will be fully skipped like in the figure 5.10.

To summarize, in the first scenario, instruction corruption (red one), instruction skip (green one), and new instruction execution (`LSLS(R1, R0, 8)`) are observed. In the second scenario, an instruction corruption occurred (red one) and an instruction is skipped (green one).

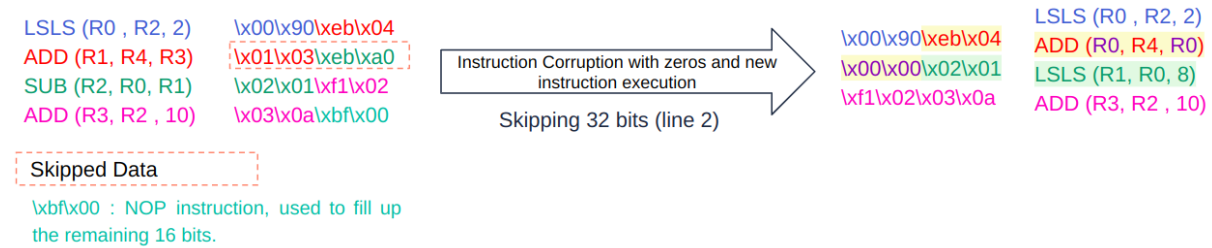


Figure 5.9: Instruction corruption with zeros - case 1.

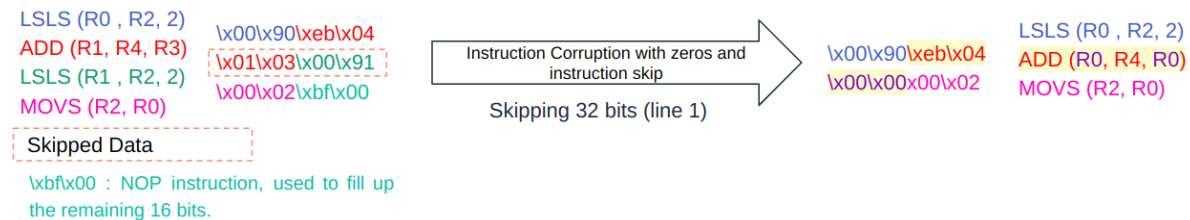


Figure 5.10: Instruction corruption with zeros fault model - case 2.

## 5.2.8 One Operand Corruption

This model contains four alternatives as demonstrated in the figure 5.11, which may be classified into two groups: successive two instructions with the same number of operands



(case (b) and (d)) or with different number of operand (case (a) and (c)).

In the first situation, as seen in the scenarios (b) and (d), each operand is corrupted with the corresponding one of the prior instruction.

In the second scenario, the destination operand is contaminated with the preceding instruction's destination operand. However, the second source operand of a 3-operand instruction is replaced with the source operand of a 2-operand instruction (case (a)); and the source operand of a 2-operand instruction is corrupted with the second source operand of a 3-operand instruction (case (c)).

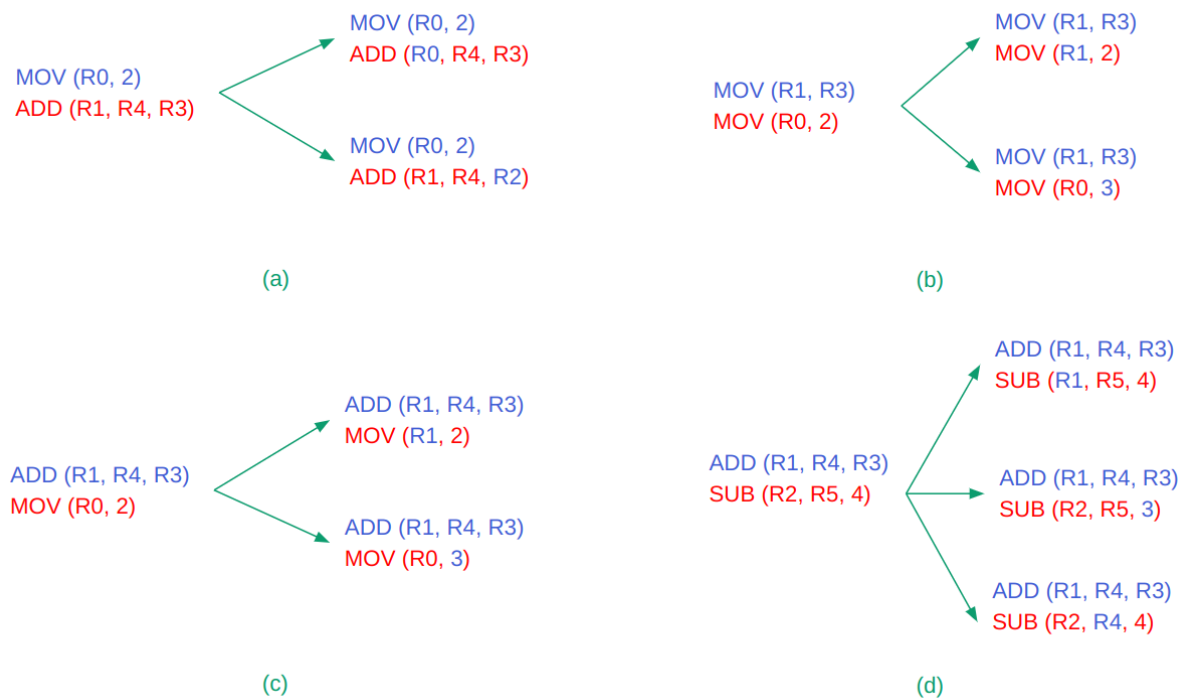


Figure 5.11: One operand corruption fault model.

### 5.2.9 One instruction corruption, changing it to MOV instruction

This model results in a corrupted instruction. Following the example given in figure 5.12, the destination operand is left unchanged while the opcode is modified by MOV and the source operand(s) are set to R0.

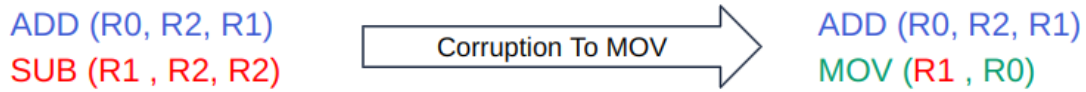


Figure 5.12: One instruction corruption, changing it to MOV instruction.

## 5.3 Automation Module

As previously said, this module is supplied through a python script called from the main one. Several CSV files are edited and compared line by line using Pandas and Numpy. It accepts as input a CSV file containing the results of fault injections, such as the one shown in the figure 5.13. Each line will be compared to the CSV file containing the simulation results of the software fault models. Each time a match is discovered, it is included to the final report. This module provides several new features, such as calculating the percentage of time each model was observed and the number of crashes for each experiment.

**Injection Outcomes**

	A	B	C	D	E	F	G
1	result1	result2	result3	result4	result5	result6	result7
2	4	4	4	4	4	4	4
3	96	96	96	96	9	96	96
4	4	4	4	4	90	4	4
5	90	90	90	90	90	90	90
6	20	20	20	9	20	1000	20
7	12	12	12	0	12	450	12
8	0	0	0	0	-86	8	8
9	9	0	0	9	9	0	0
10	0	0	0	0	0	4	4
11	12	12	12	12	12	12	12
12	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0
15	21	21	22	22	21	22	22
16	9000	15	9000	92	5	3	3

**Annotations:**

- A green bracket on the left side of the table, spanning rows 2 to 14, is labeled "R0 - R15 Registers".
- A red arrow points to the value "21" in row 15, column A, with the label "Delay".
- A yellow arrow points to the value "92" in row 16, column D, with the label "Number of times this model was observed".

Figure 5.13: Fault injection outcomes CSV file.

## 5.4 Conclusion

This chapter provided an overview of the whole process of development. It provided an explanation for the software models that were simulated at the Instruction Set Architecture (ISA) level as well as at the binary encoding level. However, it is necessary to do tests on the tool before really putting it into use. This will be the purpose of the next chapter.

## Chapter 6

# Fault injection and simulation on microprocessor architectures tests and results

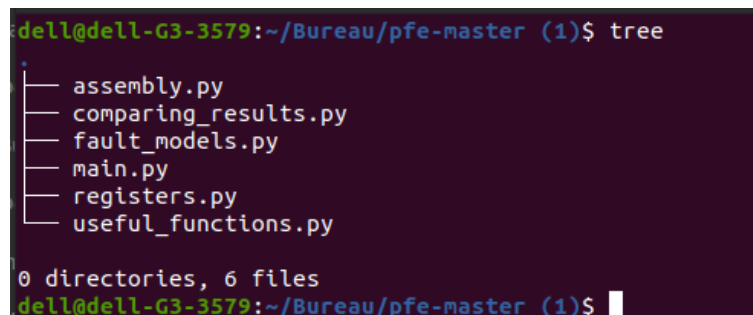
## 6.1 Introduction

In order to put the fault injection and simulation tool into test, the software tool was fed with some fault injection output to observe the analysis reported. And in the next phase, fault injections was performed in order to bypass a password check and gain an illegal access.

## 6.2 Fault Model Simulation

### 6.2.1 Getting the Source Code

In order to use the simulator, the first thing that should be done is getting its source code. This might be accomplished by either cloning the github repository or installing it as a zip file and extracting the contents<sup>1</sup>. The figure 6.1 shows the content of the project folder.



```
dell@dell-G3-3579:~/Bureau/pfe-master (1)$ tree
.
├── assembly.py
├── comparing_results.py
├── fault_models.py
├── main.py
├── registers.py
└── useful_functions.py

0 directories, 6 files
dell@dell-G3-3579:~/Bureau/pfe-master (1)$
```

Figure 6.1: Project folder's content.

### 6.2.2 Installing the requirements

In order to install the requirements the command that follows should be executed.

`pip install numpy & pip install pandas & pip install fpdf & pip install keystone-engine & pip install capstone.`

Note : it is recommended to install these requirements in a virtual environment. The commands that follow should be run prior to installing the modules for this purpose, as seen in the figure 6.2<sup>2</sup> :

`virtualenv [Name of virtual environment]`  
`source ./[Name of virtual environment]/bin/activate`

---

<sup>1</sup>Because this tool is intended to be used internally. Future research and development, however, may necessitate its eventual release to the public.

<sup>2</sup>These commands must be executed inside the project folder.

```
dell@dell-G3-3579:~/Bureau/pfe-master$ virtualenv TEST
created virtual environment CPython3.8.10.final.0-64 in 7401ms
  creator CPython3Posix(dest=/home/dell/Bureau/pfe-master/TEST, clear=False, no_
vcs_ignore=False, global=False)
  seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle
, via=copy, app_data_dir=/home/dell/.local/share/virtualenv)
    added seed packages: pip==22.1.2, setuptools==63.1.0, wheel==0.37.1
  activators BashActivator,CShellActivator,FishActivator,NushellActivator,PowerS
hellActivator,PythonActivator
dell@dell-G3-3579:~/Bureau/pfe-master$ source ./TEST/bin/activate
(TEST) dell@dell-G3-3579:~/Bureau/pfe-master$
```

Figure 6.2: Virtual environment creation.

### 6.2.3 Running the program

To use this tool, the main script must be executed. Navigate to the project folder and run the following command: `python main.py` (Fig. 6.3).

The user should next follow the instructions, which include supplying the path to the target code, to the prologue file<sup>3</sup> and deciding whether to examine the outcomes of an RTL simulation or a physical injection.

Following the successful completion of the program, two new files are created as shown in the figure 6.3: `output.csv` and `report_physical_injection.pdf`. The first one relates to the fault simulation output (the outcomes of the fault models). While the second represents the generated final report. In this scenario, the physical injection report is generated since the test was run on a physical injection result. On the other hand, if the user is interested in studying RTL injections experiments, the report will be an RTL simulation report.

```
(TEST) dell@dell-G3-3579:~/Bureau/pfe-master$ python main.py
Please enter the path to the prolog file./home/dell/PycharmProjects/Software_Models/prologue.txt

Please enter the path to the target code file./home/dell/PycharmProjects/Software_Models/target.txt
Please type R if you want to analyse RTL simulation and P if you want to analyse physical injections.P
Please enter the path holding the physical injection outcomes./home/dell/PycharmProjects/Software_Models/fault.csv
Please enter the number of experiences: 10000
(TEST) dell@dell-G3-3579:~/Bureau/pfe-master$ ls
assembly.py          output.csv          TEST
comparing_results.py __pycache__         tutorial-env
fault_models.py      registers.py        useful_functions.py
main.py              report_physical_injection.pdf
```

Figure 6.3: Successful completion of the program.

### 6.2.4 Observed results

The results of the software fault simulation will be displayed in a CSV file, as depicted in the figure 6.4, where the question mark indicates that something went wrong when computing the specific model and might be the original cause of a crash observed in the hardware.

<sup>3</sup>Initialization file : All registers should be initialized using the MOV instruction.

initial	Golden	skip_1_0	skip_1_1	skip_1_2	skip_1_3	skip_1_4	skip_1_5	skip_1_6	skip_2_0	skip_2_1	skip_2_2	skip_2_3	skip_2_4	skip_2_5	sk
1073809408	1073809408	1073809408	1073809408	1073809408	1073809408	1073809408	?	1073809408	1073809408	1073809408	1073809408	1073809408	1073809408	1073809408	1073809408
9	15	15	15	9	15	15	?	15	15	9	9	15	15	15	9
4096	134217728	134217728	4096	134217728	134217728	134217728	?	134217728	4096	4096	134217728	134217728	134217728	134217728	4096
90	115	115	115	115	105	115	?	100	115	115	105	105	115	100	115
20	31	31	31	31	31	20	?	31	31	31	31	20	20	31	31
12	112	112	112	112	102	112	?	112	112	112	102	102	12	12	11
12	12	12	12	12	12	12	?	12	12	12	12	12	12	12	11
536878360	536878360	536878360	536878360	536878360	536878360	536878360	?	536878360	536878360	536878360	536878360	536878360	536878360	536878360	536878360
0	20	0	20	20	20	20	?	20	0	20	20	20	20	20	0
12	12	12	12	12	12	12	?	12	12	12	12	12	12	12	11
536871168	536871168	536871168	536871168	536871168	536871168	536871168	?	536871168	536871168	536871168	536871168	536871168	536871168	536871168	536871168
0	0	0	0	0	0	0	?	0	0	0	0	0	0	0	0
536878360	536878360	536878360	536878360	536878360	536878360	536878360	?	536878360	536878360	536878360	536878360	536878360	536878360	536878360	536878360

Figure 6.4: Software fault simulation outcomes.

The final report is shown in the figure 6.5. As illustrated, the results will be categorized according to their delay<sup>4</sup>. Each line of the physical injection result will be presented, followed by the number of models that match. The models will then be listed with a detailed description. The model's percentage of appearance throughout the trial will be displayed enabling countermeasure designers to identify which models are more likely to take place and subsequently provide optimized countermeasures. Finally, the total number of crashes will be presented for each delay.

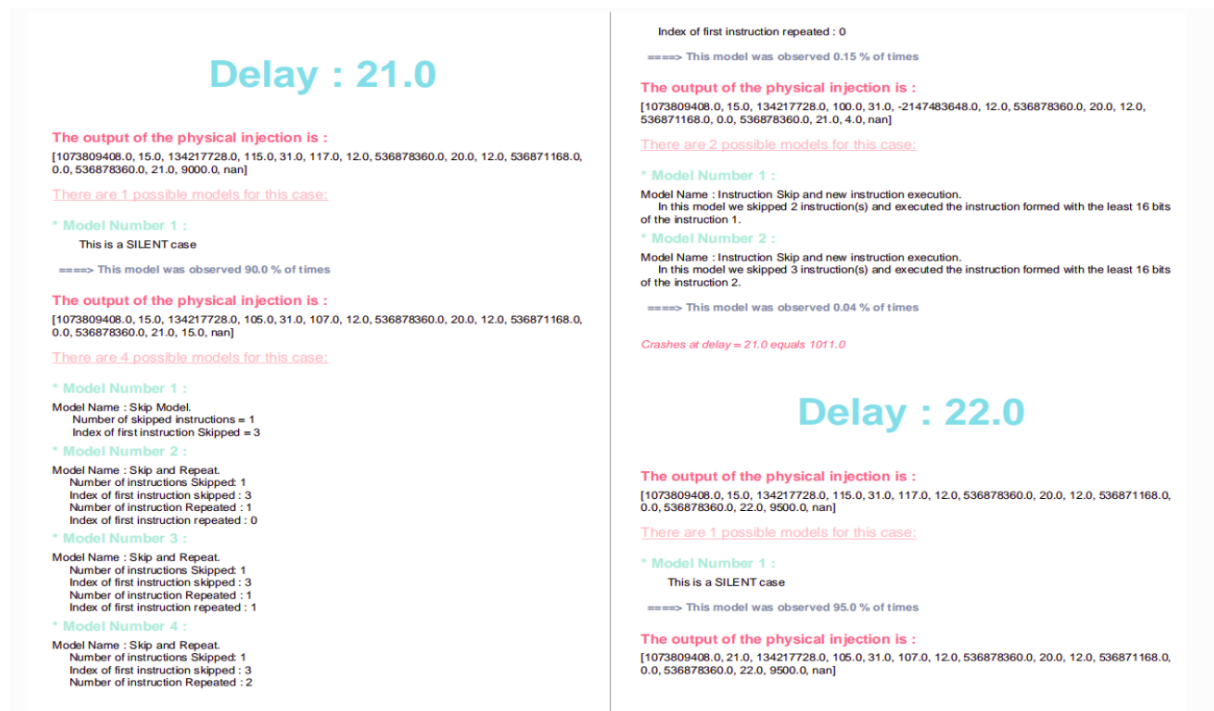


Figure 6.5: Generated report.

<sup>4</sup>the time elapsed between the rising edge of the synchronization trigger signal and the rising edge of the intended clock cycle.

## 6.3 Physical fault injection

### 6.3.1 Clock glitch

During the fault injection experiment, an effective and widely utilized fault injection approach is used: clock glitch. It is accomplished by inserting a glitch into the regular clock cycle, causing one or more flip-flops to accept the incorrect state and therefore change the command (Fig. 6.6). In fact, one instruction will be fetched, another will be decoded, and a third will be executed at each rising edge of a clock<sup>5</sup>. Therefore, the introduced glitch would make it seem as if a new cycle had begun. This forces the CPU to retrieve a new instruction, decode the prior one, and then execute the previously decoded instruction.

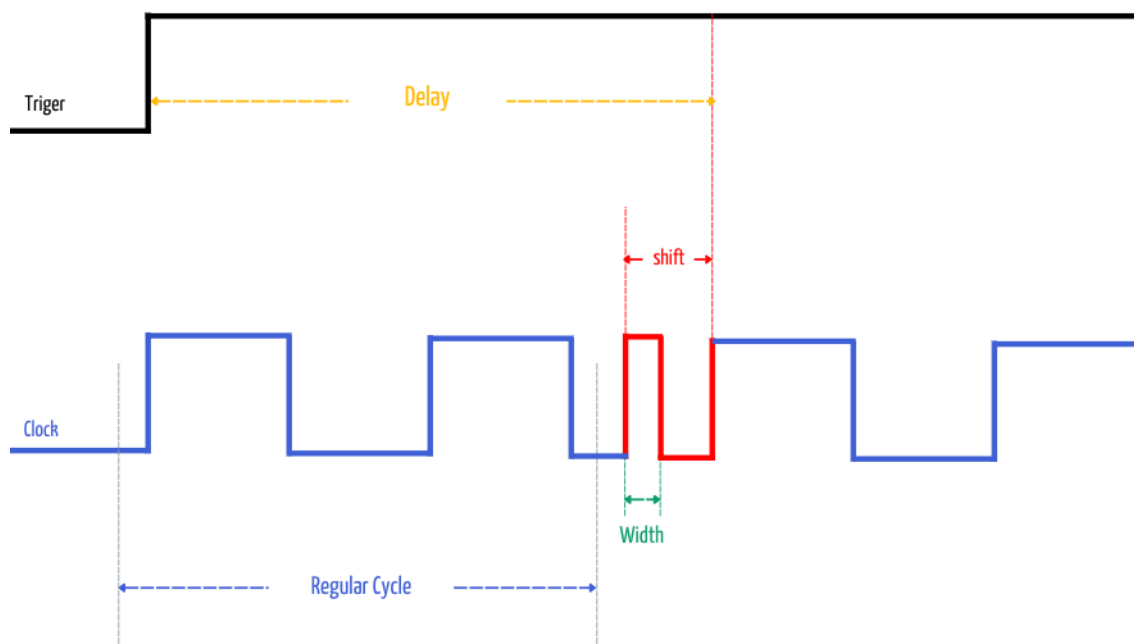


Figure 6.6: Clock glitch.

However, due to the fact that the glitch interferes with the usual behavior of the clock signal, the processor would not have sufficient time to effectively handle the instructions, which would result in a variety of erroneous behaviors.

Three parameters define the clock glitch and must be appropriately specified in order to inject a fault [1]:

- Delay: the time elapsed between the rising edge of the synchronization trigger signal and the rising edge of the desired clock cycle.
- Shift: the elapsed time between the glitch's rising edge and the start of the desired clock cycle.
- Width: the length of the glitch.

<sup>5</sup>Refer to figure 2.10 : instruction cycle.



### 6.3.2 ChipWhisperer

The experiment is carried out using a ChipWhisperer board [14]. This open source toolchain offers a dedicated environment for testing the devices' susceptibility to side channel attacks.

It is not the goal board in and of itself. It's a single-board solution that measures high-speed power, has a built-in target and its programmer, and may be used as a platform to carry out fault injection. This makes it an excellent instrument for fault injection and side channel investigation.

When it comes to ChipWhisperer, there are two main points of interest. First, there is the power analysis, in which an attacker uses the information supplied by the device's power usage to launch an attack. Second, clock glitching and voltage attacks, which momentarily disrupt a device's clock or power to cause undesirable behavior (such as skipping a password check).

This base board is linked to the control Computer through a USB connection.



Figure 6.7: ChipWhisperer Logo.

### 6.3.3 Target

#### Hardware

In this experiment, the STM32F303 microcontroller serves as the target board. Embedded inside its hardware is the ARM Cortex-M4 processor architecture.

#### Software

The target software implementation in question is a PIN (Personal Identification Number) verification system called "VerifyPIN" (Fig. 6.8). This algorithm is widely used in the field of hardware protection. This method is used to authenticate users to a device as a password. A system-implemented reference value is compared to a user's given PIN to determine whether or not the user should be granted access. These passphrases are often just 4 or 5 digits long, therefore their security is not dependent on their resistance to brute force attacks, instead by limiting the number of trials. In order to get access, the system's behavior should be altered such that the access is granted even if an incorrect code is supplied. This is accomplished by bypassing the authentication instructions.

```

char passwd[] = "touch";
char passok = 1;
int cnt;

trigger_high();

//Simple test - doesn't check for too-long password!
for(cnt = 0; cnt < 5; cnt++){
    if (pw[cnt] != passwd[cnt]){
        passok = 0;
    }
}

trigger_low();

simpleserial_put('r', 1, (uint8_t*)&passok);
return 0x00;

```

Figure 6.8: Target code.

### 6.3.4 Environment setup

To use the ChipWhisperer to its full potential, it must first be installed on the control computer<sup>6</sup>. Once launched, a Jupyter notebook interface is presented in the browser. First, the toolchain's specified target code should be generated, and a connection to the ChipWhisperer board should be established (Fig. 6.9).

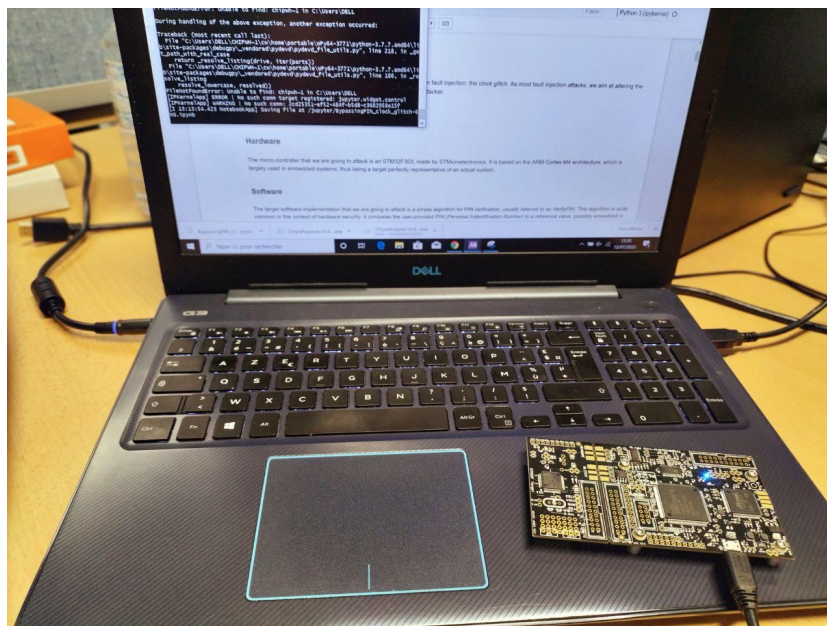


Figure 6.9: ChipWhisperer board setup.

<sup>6</sup><https://chipwhisperer.readthedocs.io/en/latest/index.html>

Then, the just compiled firmware must be uploaded to the board, where it will be written in the microcontroller's Flash memory. The card is then reset, allowing the microcontroller to load the relevant firmware and begin execution (Fig. 6.10).

```
Entrée [11]: import chipwhisperer as cw
fw_path = "{}hardware/victims/firmware/simpleserial-glitch/simpleserial-glitch-{}.hex".format(CW_PATH, PLATFORM)
cw.program_target(scope, prog, fw_path)
scope.io.nrst = False
time.sleep(0.05)
scope.io.nrst = "high_z"
time.sleep(0.05)
target.flush()

Detected unknown STM32F ID: 0x446
Extended erase (0x44), this can take ten seconds or more
Attempting to program 5475 bytes at 0x8000000
STM32F Programming flash...
STM32F Reading flash...
Verified flash OK, 5475 bytes
```

Figure 6.10: Setting up and programming the target.

### 6.3.5 Testing the normal behaviour

Before proceeding with the injection, it is essential to ensure the loaded PIN verification system firmware is functioning correctly. It should accept the accurate PIN (Fig. 6.11) but reject the incorrect one (Fig. 6.12).

```
Entrée [12]: pw = "aeiou".encode('ascii')
target.simpleserial_write('p', pw)
val = target.simpleserial_read_witherrors('r', 1, glitch_timeout=10)
passok = int.from_bytes(val["payload"], byteorder='big', signed=False)
print(val)
print(passok)
if not(passok):
    print("FAILED")
else:
    print("SUCCESS")

{'valid': True, 'payload': bytearray(b'\x00'), 'full_response': 'r00\n', 'rv': 0}
0
FAILED
```

Figure 6.11: Testing the normal behaviour : Wrong password.

#### Correct password

We can use the same code to send now the correct password correct ("touch"), as shown below. The returned value should be now `rv=1`, as it can be checked through the appearance of the message `SUCCESS`.

```
Entrée [13]: pw = "touch".encode('ascii')
target.simpleserial_write('p', pw)
val = target.simpleserial_read_witherrors('r', 1, glitch_timeout=10)
passok = int.from_bytes(val["payload"], byteorder='big', signed=False)
print(val)
print(passok)
if not(passok):
    print("FAILED")
else:
    print("SUCCESS")

{'valid': True, 'payload': bytearray(b'\x01'), 'full_response': 'r01\n', 'rv': 0}
1
SUCCESS
```

Figure 6.12: Testing the normal behaviour : correct password.

### 6.3.6 Clock glitch parameters

Once the intended behavior of the code has been verified, fault injection may be conducted to alter it. The width, offset, and delay are the three definable parameters of a clock glitch.

Initially, there is no straightforward method to predict the optimal right set of values that will permit injecting an exploitable defect. This means that all possible combinations of values need to be investigated before the best one can be determined. This requires testing all 970200 possible combinations, where the width ranges from 1 and 49, the offset is between -49 and 49, and the delay varies from 1 and 200. Although this could be the only option sometimes, it is known to be time consuming. As a result, a few strategies are used for speeding up the process by lowering the freedom degrees and hence the amount of dimensions to investigate. The three parameters are independent. The delay, in instance, is mostly determined by the application being targeted, whilst width and offset are determined by the platform's real physical properties. The easy known aspect of injecting faults in a for loop and so the two nested loop to determine the set of offset and width leading to an exploitable fault was used. In fact, a characterisation algorithm made of two nested for loops, each with 50 iterations, that each increase a global counter as shown in figure 6.13 is used.

```

reboot_flush()

resets = [] # List to save those parameters leading to a reset
glitches = [] # List to save those parameters leading to an actual (useful) glitch

largeurs = [*range(1,49,1)] # TO BE COMPLETED
decalages = [*range(-49,49,1)] # TO BE COMPLETED
delais = [*range(1,200,20)] # TO BE COMPLETED

for scope.glitch.width in largeurs:
    print("Width : {}".format(scope.glitch.width))
    for scope.glitch.offset in decalage:
        for scope.glitch.ext_offset in delais:
            for repetition in range(3):
                scope.arm()

                target.write("g\n")
                ret = scope.capture()
                val = target.simpleserial_read_witherrors('r', 4, glitch_timeout=10)

                if ret: #here the trigger never went high - sometimes the target is stil crashed from a previous glitch
                    resets.append((scope.glitch.width, scope.glitch.offset))
                    reboot_flush()
                elif val["payload"]:
                    loop_counter = int.from_bytes(val["payload"], byteorder='little')
                    if loop_counter != 2500:
                        glitches.append((scope.glitch.width, scope.glitch.offset))

```

Figure 6.13: Characterization code.

A series of clock glitches are introduced in order to relieve the restriction on the delay value, which is required in order to target a particular instruction. In this case, the delay may advance by 20 (if the glitch is repeated 20 times), rather than 1, resulting in 10 possibilities rather than 200. The figure 6.14 identifies the width and offset leading to a successful exploit.

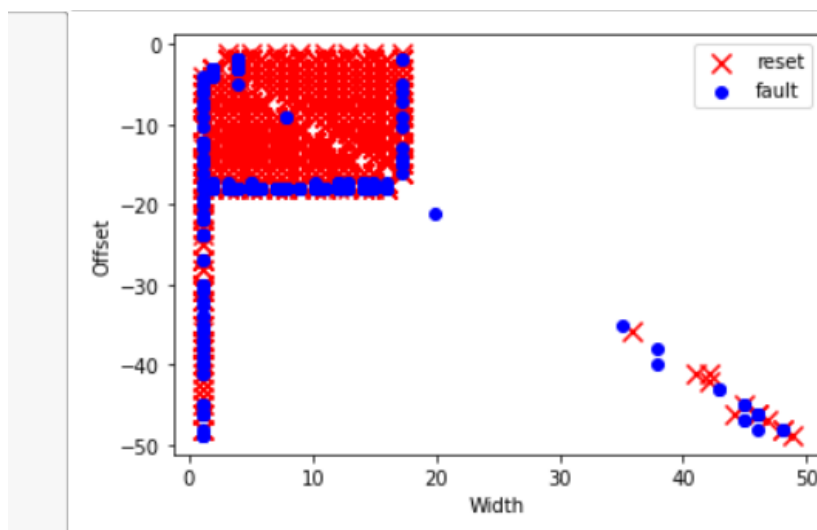


Figure 6.14: Offset and width pair leading to a fault.

After determining a suitable offset and width for the board in question, the delay parameter should be defined. This value will be determined in the next step, since it is software-specific.

### 6.3.7 Attacking VerifyPin

Successfully attacking the VerifyPIN requires knowing the delay parameter. Therefore for the sets of offset and width determined in the previous task, a loop for all possible values of delay is defined (Fig. 6.15). In order to circumvent the password verification, we will repeatedly try "aeiou" as an invalid authentication password instead of "touch" for each possible value of delay. If our authentication is successful, the value of the variable "passok" will be set to one (Fig. 6.8), and as a result, the text "CASSE!" will be printed. As can be seen in Figure 6.15, we were able to bypass a password check and get authorized using the wrong password.

```

for scope.glitch.width in largeurs:
    for scope.glitch.offset in decalages:
        for scope.glitch.ext_offset in delais:
            #pw = "touch".encode('ascii') # Correct password
            pw = "aeiou".encode('ascii') # Wrong password

            for repetition in range(10):
                scope.arm()

                target.simpleserial_write('p', pw)
                ret = scope.capture()
                val = target.simpleserial_read_witherrors('r', 1, glitch_timeout=20)
                #print(val)
                #print(passok)
                if ret: #here the trigger never went high - sometimes the target is stil crashed from a previous glitch
                    resets.append(scope.glitch.ext_offset)
                    reboot_flush()
                elif val["valid"]:
                    passok = int.from_bytes(val["payload"], byteorder='big', signed=False)
                    if (passok):
                        print("CASSE!")

```

CASSE!

Figure 6.15: Attacking VerifyPin.

### 6.3.8 Understanding the fault

The experiment was based on a real-case scenario in which an attacker may use fault injection to evade a password check. Therefore, for the sake of simplicity, the example was carried out using C code rather than assembly code. Nevertheless, simpler assembly target code is often utilized for analysis purpose.

In order to have a better insight of how the fault was performed, let's start the demonstration with a simple for loop.

#### Potential for loop flaws

The figure 6.16 demonstrates a basic for loop with assembly instructions. R1 will be initialized to zero and then increased by one at each iteration of the for loop until being greater than three.

In the context of the fault injection that was performed, we will suppose that one instruction is skipped. Different effects may be produced based on the particular targeted instruction:

- **ADD R1, 1:** if this instruction is skipped, the counter will not be increased; therefore, an additional iteration of the loop will be performed. In other terms, the loop will be executed five times, rather than the usual four.
- **CMP R1, 3:** skipping this instruction prevents the counter from being compared to the reference value. The flags in the processor (indicating whether or not further jumps are required) will not be updated, affecting looping. For instance, the fault will be ineffective in the second iteration when R1 equals two. However, if R1 is four, the flags will not be updated and one more iteration will be performed. Additional consequences may result from the processor's design.
- **BLE debut boucle:** If this instruction is skipped, the execution will not return to the beginning of the loop regardless of the counter's value, even if it is still less than the given limit. Therefore, the loop will end sooner than anticipated. As a result, the proposed fault model is considered as highly effective and enable significant modification to the behavior of an algorithm.

```
MOV R1, 0          // Loop counter initialisation
start_loop:        // Label
...                // Loop body
ADD R1, 1          // Counter increment
CMP R1, 3          // Comparing counter to reference
BLE start_loop     // Jump to label if counter is less or equal to boundary (Branch Less or Equal)
```

Figure 6.16: For Loop.

## Real case scenario

The previous paragraph covered the fundamentals of a simple for loop as well as the ramifications of skipping each of its instructions. It was determined that bypassing the brunch instruction is the most appealing for skipping the pin verification instructions. Let's put this theory to the test by examining the actual case scenario. The assembly instructions of the authentication function are shown in Figure 6.17 (right section).

For the sake of clarity, the code is separated into sections, and the most important events will be highlighted.

- Line 35: Bytes of memory are created and initialized with the string holding the correct password "touch."
- Lines 37 - 43 : Green block : A particular address memory is saved in the register R7 as a reference.
- Lines 44 - 50 : Yellow block : The correct password "touch" is stored at the address [R7+20].
- Lines 51 - 52 : Blue block : The variable "passok" is initialized to one and saved at the address [R7+19]. This variable authenticates the user, hence the objective is to maintain it to one regardless of the provided password.
- Lines 54 - 56 [...] 72 - 74 [...] 76 - 78 : Purple block : It indicates the for loop being targeted.

Lines 54 - 56 : Setting the counter to zero and jumping to L6.

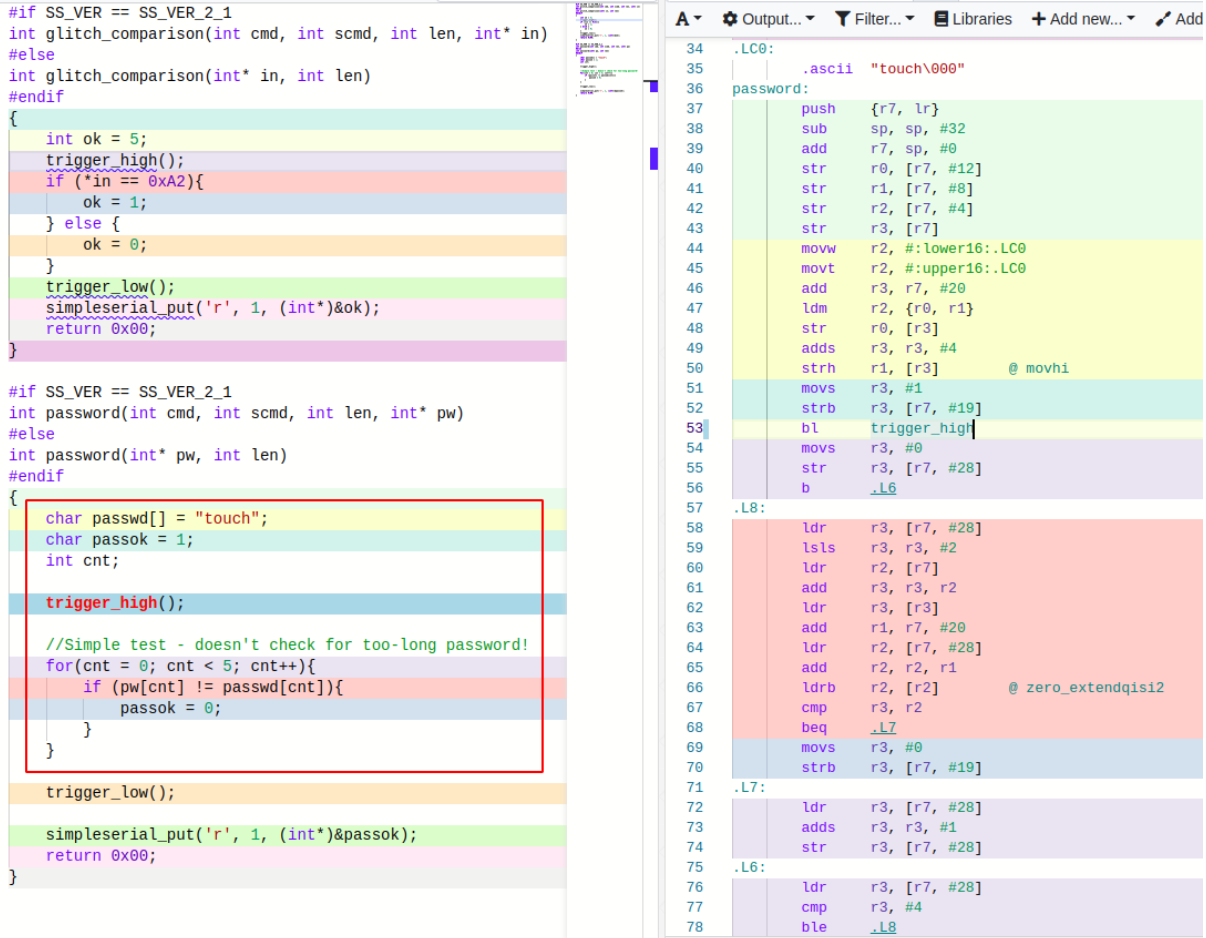
Lines 74 - 78 : The L6 section checks whether the counter is less than or equal to four and, if so, jumps to L8; otherwise, the program terminates.

Lines 72 - 74 : If the checked character of the suggested password is similar to the correct one, the L7 section increments the counter by one.

- Lines 58 - 68 : Red Block : The L8 section performs a character-based comparison between the suggested password and the valid one. If a match was found, the program will jump to L7 to increase the counter; otherwise, the "passok" variable will be set to zero and the user will not be authenticated.

Putting it all together: the targeted function will reset a counter to zero and then check to see whether the suggested password character at this position is identical to the correct one. If this is the case, the counter is increased and the following three characters are checked in the same manner. If this is not the case, the "passok" variable is set to zero, indicating that the suggested password does not match the correct one. This loop is terminated whenever an incorrect character is identified or the counter reaches five.





```

#if SS_VER == SS_VER_2_1
int glitch_comparison(int cmd, int scmd, int len, int* in)
#else
int glitch_comparison(int* in, int len)
#endif
{
    int ok = 5;
    trigger_high();
    if (*in == 0xA2){
        ok = 1;
    } else {
        ok = 0;
    }
    trigger_low();
    simpleserial_put('r', 1, (int*)&ok);
    return 0x00;
}

#if SS_VER == SS_VER_2_1
int password(int cmd, int scmd, int len, int* pw)
#else
int password(int* pw, int len)
#endif
{
    char passwd[] = "touch";
    char passok = 1;
    int cnt;

    trigger_high();

    //Simple test - doesn't check for too-long password!
    for(cnt = 0; cnt < 5; cnt++){
        if (pw[cnt] != passwd[cnt]){
            passok = 0;
        }
    }

    trigger_low();

    simpleserial_put('r', 1, (int*)&passok);
    return 0x00;
}

```

```

34 .LC0:
35 .ascii "touch\000"
36 password:
37     push    {r7, lr}
38     sub     sp, sp, #32
39     add     r7, sp, #0
40     str     r0, [r7, #12]
41     str     r1, [r7, #8]
42     str     r2, [r7, #4]
43     str     r3, [r7]
44     movw    r2, #:lower16:.LC0
45     movt    r2, #:upper16:.LC0
46     add     r3, r7, #20
47     ldm     r2, {r0, r1}
48     str     r0, [r3]
49     adds    r3, r3, #4
50     strh    r1, [r3]      @ movh1
51     movs    r3, #1
52     strb    r3, [r7, #19]
53     bl      trigger_high
54     movs    r3, #0
55     str     r3, [r7, #28]
56     b       .L6
57 .L8:
58     ldr     r3, [r7, #28]
59     lsls    r3, r3, #2
60     ldr     r2, [r7]
61     add     r3, r3, r2
62     ldr     r3, [r3]
63     add     r1, r7, #20
64     ldr     r2, [r7, #28]
65     add     r2, r2, r1
66     ldrb    r2, [r2]      @ zero_extendqis12
67     cmp     r3, r2
68     beq     .L7
69     movs    r3, #0
70     strb    r3, [r7, #19]
71 .L7:
72     ldr     r3, [r7, #28]
73     adds    r3, r3, #1
74     str     r3, [r7, #28]
75 .L6:
76     ldr     r3, [r7, #28]
77     cmp     r3, #4
78     ble     .L8

```

Figure 6.17: Target software - Assembly code.

To fulfill the desired goal of getting authorized with an incorrect password, the red block must be bypassed, in other terms the check procedure will be skipped.

The first potential method involves targeting line 56. In this scenario, The counter will be set to 0, but the program will exit immediately before beginning the loop.

The second alternative method is to skip line 78. In this scenario, the loop is initialized by determining if the counter is less than five, the program exists before beginning the first iteration.

It is possible that other, more complex models might lead to the same desired goal. So, to have a better understanding of the exploited vulnerability, it is necessary to extract register values. Then, all potential fault models should be simulated, and a comparison of the results will reveal which models the target system is vulnerable to and so, more effective countermeasures can be developed.

## 6.4 Conclusion

In this chapter, we made sure that our tool was functioning correctly in all aspects. Taking advantage of the output of the fault injection and then submitting it to the software tool,



the outcomes of each software model were manually inspected, as was the report that was ultimately generated. After finishing this step, the tool successfully completed all of the tests and is now ready to be used.

## General conclusion

## General conclusion

Within the context of a graduate internship project, we worked on identifying fault injection vulnerabilities in microprocessor architectures. For instance, sophisticated fault injections effects are modeled in the software layer while keeping a comparative link between the three layers : RTL , ISA and hardware. This approach helps in understanding the effects of each attack and the vulnerability being exploited.

The recommended solution is a Linux-based Python script. It is designed for usage on 32-bit microcontrollers and tested on processors that support the THUMB2 instruction set (ISA). The project introduces an automated method for software model simulation and fault injection outcomes analysis. In the first step, it takes into consideration the targeted code and the initial state of registers to abstract the lowest levels of fault injection layers. It simulates realistic and effective models using both existing and new models inferred from physical fault injections. In the second step, the previously acquired results from the software model simulation will be compared to those of physical injection and RTL simulation. This work will aid in the development of adequate and effective countermeasures, since developers will be able to assess which models can be executed on the target board they want to test and the likelihood of the injection success.

As a significant enhancement to our technology, more architectures might be supported. In addition, the simulator might be expanded to include software models that are afterwards observed. In conclusion, a substantial update would include the consideration of a range of user input and the generation of several reports, all of which would be compared to one another before providing the final report.

# Bibliography

1. Alshaer, I., Colombier, B., Deleuze, C., Beroulle, V. & Maistri, P. *Microarchitecture-aware Fault Models: Experimental Evidence and Cross-Layer Inference Methodology in 2021 16th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)* (2021), 1–6.
2. Proy, J., Heydemann, K., Berzati, A., Majéric, F. & Cohen, A. *A first ISA-level characterization of EM pulse effects on superscalar microarchitectures: a secure software perspective in Proceedings of the 14th International Conference on Availability, Reliability and Security* (2019), 1–10.
3. *GitHub - Riscure FiSim: An open-source deterministic fault attack simulator prototype* [Online]. <https://github.com/Riscure/FiSim>.
4. Troughkine, T. *SoC physical security evaluation* PhD thesis (Université Grenoble Alpes [2020-....], 2021).
5. Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B. & Encrenaz, E. *Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller in 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography* (2013), 77–88.
6. Berthier, M. *et al. Idea: embedded fault injection simulator on smartcard in International Symposium on Engineering Secure Software and Systems* (2014), 222–229.
7. Ziade, H., Ayoubi, R. A., Velazco, R., *et al.* A survey on fault injection techniques. *Int. Arab J. Inf. Technol.* **1**, 171–186 (2004).
8. Alshaer, I., Colombier, B., Deleuze, C., Beroulle, V. & Maistri, P. *Variable-Length Instruction Set: Feature or Bug?* in *2022 Euromicro conference on Digital System Design (DSD)* (2022), 1–7.
9. ARM. *The Thumb instruction set* [Online]. <https://developer.arm.com/documentation/ddi0210/c/CACBCAAE>.
10. ARM. *About Thumb-2* [Online]. <https://developer.arm.com/documentation/ddi0210/c/CACBCAAE>.
11. *Understanding ARM Architectures* [Online]. <https://www.informit.com/articles/article.aspx?p=1620207&seqNum=3>.
12. *Cortex M3 - Technical Reference Manual* [Online]. [https://www.keil.com/dd/docs/datashts/arm/cortex\\_m3/r1p1/ddi0337e\\_cortex\\_m3\\_r1p1\\_trm.pdf](https://www.keil.com/dd/docs/datashts/arm/cortex_m3/r1p1/ddi0337e_cortex_m3_r1p1_trm.pdf).
13. Riviere, L. *et al.* *High precision fault injections on the instruction cache of ARMv7-M architectures in 2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* (2015), 62–67.

14. O'flynn, C. & Chen, Z. D. *Chipwhisperer: An open-source platform for hardware embedded security research* in *International Workshop on Constructive Side-Channel Analysis and Secure Design* (2014), 243–260.