Formal fault injection vulnerability detection in binaries : a software process and hardware validation

Nisrine Jafri

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE **U**NIVERSITE **B**RETAGNE **L**OIRE

Ecole Doctorale N°601
Mathèmatique et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par

« Nisrine JAFRI »

« Formal Fault Injection Vulnerability Detection in Binaries »

«A Software Process and Hardware Validation»

Thèse présentée et soutenue à RENNES , le 25 mars 2019
Unité de recherche : Inria, TAMIS team
Thèse N° :

Rapporteurs avant soutenance :
Marie-Laure POTET, Professeur des Universités, ENSIMAG
Jean-Yves MARION, Professeur des Universités, Université de Lorraine

Composition du jury :

Président : Pierre-Alain FOUQUE, Professeur des Universités, Université Rennes 1
Examinateurs :
Leijla BATINA, Professeur des Université s, Université Radboud de Nimegue
Shivam BHASIN, Chercheur, Université technologique de Singapour
Sylvain GUILLEY, Professeur des Universités, Télécom ParisTech
Annelie HEUSER, Chercheur, CNRS

Dir. de thèse : Jean-Louis LANET, Chercheur, Inria
Co-dir. de thèse : Axel LEGAY , Professeur des Universités, Université catholique de Louvain

Invité(s)

"The summit of happiness is reached when a person is ready to be what he is."

Desiderius Erasmus

إلى من بالحب غمروني وبجميل السجايا أدبوني

إلى أ ّمي و أبي

UNIVERSITY RENNES 1

Abstract

Inria

Doctoral School MATHSTIC

Doctor of Philosophy

Formal Fault Injection Vulnerability Detection in Binaries - A Software Process and Hardware Validation

by Nisrine JAFRI

Fault injection is a well known method to test the robustness and security vulnerabilities of systems. Detecting fault injection vulnerabilities has been approached with a variety of different but limited methods. Software-based and hardware-based approaches have both been used to detect fault injection vulnerabilities. Software-based approaches can provide broad and rapid coverage, but may not correlate with genuine hardware vulnerabilities. Hardware-based approaches are indisputable in their results, but rely upon expensive expert knowledge, manual testing, and can not confirm what fault model represent the created effect.

First, this thesis focuses on the software-based approach and proposes a general process that uses model checking to detect fault injection vulnerabilities in binaries. The efficacy and scalability of this process is demonstrated by detecting vulnerabilities in different cryptographic real-world implementations.

Then, this thesis bridges software-based and hardware-based fault injection vulnerability detection by contrasting results of the two approaches. This demonstrates that: not all software-based vulnerabilities can be reproduced in hardware; prior conjectures on the fault model for electromagnetic pulse attacks may not be accurate; and that there is a relationship between software-based and hardware-based approaches. Further, combining both software-based and hardware-based approaches can yield a vastly more accurate and efficient approach to detect genuine fault injection vulnerabilities.

Keywords: Fault Injection, Vulnerability Detection, Model Checking, Formal Methods

Acknowledgements

I would start by expressing my gratitude to my thesis directors Jean-Louis Lanet and Axel Legay. Thank you for giving me the opportunity to experience the PhD journey. Thanks for your directions through the path, your advices and comments.

My sincere thanks to Marie-Laure Potet and Jean-Yves Marion for accepting to review my work. Marie-Laure Potet, thank you for your advices and goodwill. Jean-Yves Marion, thank you for your availability and pertinent remarque.

Thanks for the jury members, Leijla Batina, Shivam Bhasin, Pierre-Alain Fouque, Sylvain Guilley for accepting to be part of my jury.

Special thanks to my supervisor Thomas Given-Wilson. My research would have been impossible without your aid, support, and encouragement. Thanks for all the shared techniques and tips that I will carry on with me, and share as well.

I owe a very important debt to Annelie Heuser, thanks for helping me finalise this manuscript, but also for all your support and encouragement, it was a real pleasure working with you.

Thanks to Clementine Maurice my monitor, your advices, suggestion and encouragement were a great help to me.

I would also like to thank Ronan Lashermes and Sebanjila Kevin Bukasa from the LHS lab for their help in conducting the hardware experiments.

Heartfelt thanks go to all the tamis team members. A special thank to Olivier Zendra the team leader for his support and help in order to defend my thesis. A unique thanks to Cecile Bouton. Thank you for your goodwill and support, you have been a second mother to me. thanks to the members who become true friends to me. Alex and Kevin we will remain the Trio of Porto. Tania, thanks for everything but especially for the free hugs. Cassius, thanks for your positive energy, and all the shared information about the music bands. Delphine, it was a pleasure to have you as an officemate. Thanks to all the team members with whom I shared a great moment: Stefano, Ioana, Lamine, Céline, Yoann, Christophe, Laurent, Leo, Ludo, Jean, Hélène, Florian, Fabrizio ...

I am profoundly grateful to all my friends for their emotional support through the way, thanks Meriem, Fati, Mouad, Routa, Rahaf, Farah. A special thanks to Pierre-Yves who become more than a friend. Thanks for your support, through the last miles of this journey you had the right words to boost my motivation.

Thanks to all my Spanish, Sewing, Swimming, and Salsa teachers, spending time in your courses helped me having a balanced life.

Thanks to my second family in Rennes, Gabrielle and Claire. You've been a true family to me, you embraced me with your love and care, and made me feel at home.

And lastly, all the thank goes to my beloved family for their unlimited love and support. Thanks to my parents and brothers Youssef and Youness. My grandparents and to all members of JAFRI and Chouka families, for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Nisrine JAFRI

Contents

List of Figures

List of Tables

Résumé en français

L'injection de faute est une technique utilisée pour attaquer les systèmes mais aussi pour évaluer leur robustesse. Le but d'une injection de faute est d'induire un effet spécifique au niveau du matériel créant une erreur exploitable au niveau du logiciel.

Actuellement de plus en plus de systèmes opèrent dans des milieux hostiles ce qui les rend vulnérables à toutes sortes d'attaques, d'où la nécessité d'évaluer leur robustesse face aux attaques par injection de fautes.

L'objectif de cette thèse est d'étudier l'effet d'une telle faute matérielle sur le logiciel, et voir si un tel effet peut créer une vulnérabilité au niveau logiciel. En premier lieu nos recherches visaient à explorer les différentes approches logicielles existantes qui étaient limitées à certains types de vulnérabilités et de modèles de faute. La première contribution de cette thèse est donc une approche automatisée qui utilise des techniques de vérification formelle pour la détection de vulnérabilité intitulé FIVD (Fault Injection Vulnerability Detection process). En outre la vérification est faite au niveau binaire ce qui représente le mieux la majorité des attaques et qui ne limite pas le type de modèle de faute utilisé pour la simulation.

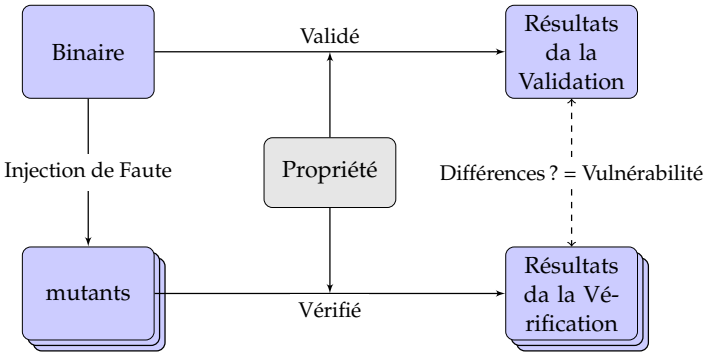

FIGURE 1 – Première Contribution : Le processus FIVD

L'efficacité de cette approche a été montrée en l'appliquant à des algorithmes de cryptographie (PRESENT, SPECK) implémentés dans les systèmes embarqués. Les résultats des expériences ont montré qu'en utilisant l'approche logicielle, il était possible de détecter différents types de vulnérabilités, des vulnérabilités déjà connues dans la littérature mais aussi des vulnérabilités nouvelles.

En deuxième lieu, et vu que l'approche logicielle n'est pas suffisante pour confirmer que la vulnérabilité détectée correspond à une vulnérabilité réelle qui peut être créée physiquement, il a été nécessaire d'explorer aussi l'approche physique. La deuxième contribution de cette thèse est donc la combinaison des deux approches (logicielles et physiques) afin d'explorer de nouvelles méthodes de détection de vulnérabilité par injection de fautes.

Les expériences réalisées ont montré que les résultats des deux approches coïncident mais ne sont pas totalement identiques. L'approche physique permet la détection de vulnérabilités réelles mais c'est une approche très couteuse qui nécessite beaucoup

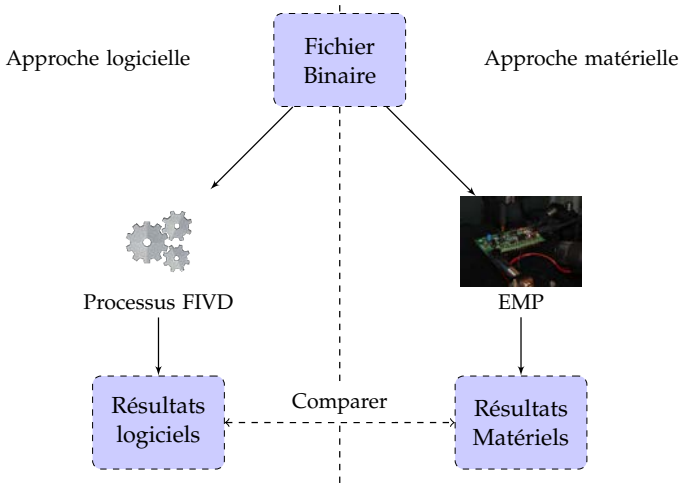

FIGURE 2 – Seconde Contribution : la combinaison de l'approche logicielle et physique

d'expertise et de temps. L'approche logicielle est moins couteuse et nécessite moins d'efforts et de temps mais les résultats de l'approche logicielle n'étaient pas tous validés par les expériences physiques. Afin de limiter le coût, le temps, et l'effort nous proposons la combinaison des deux approches de la manière suivante. Utiliser l'approche logicielle pour détecter les endroits les plus susceptibles d'être vulnérables dans le système à vérifier, et après utiliser l'approche physique pour tester directement ces endroits vulnérables sans avoir à analyser tout le système.

Introduction and Background

Chapter 1

Introduction

This chapter gives an introduction to this thesis. It first presents the context and motivation behind the subject of this thesis. Next, it shows the contributions of this thesis. Then, it lists the papers published during this thesis. Finally, it describes how the rest of this thesis is organised.

Sommaire

1.1 Context and Motivation

Embedded systems are becoming "unavoidable" in many domains, e.g., health care, aerospace, transportation, and energy, where they manipulate private and sensitive data, and perform mission critical tasks. In the health care domain, the manipulated data is vital to a patient's life, any flaw in the system may cause death. In the aerospace domain, the systems must be designed to operate in space, where the temperature and environment can be fatal to system hardware. In the transportation domain, a flaw in a flight system can cause the death of hundreds of people.

The majority of embedded systems operate in hostile environments, where an embedded system's hardware may be disrupted. Embedded system disturbance can be unintentional (e.g. background radiation, power interruption [12, 90]) or intentional (e.g. induced electromagnetic pulse (EMP) [41, 96], rowhammer [113, 127, 150]).

Unintentional disturbance is generally attributed to the environment [49, 90]. An example of this is one of the first observed fault injections where radioactive elements present in packing materials caused bits to flip in chips [12].

Intentional disturbance occurs when the injection is done by an *attacker* with the intention of changing program execution [96, 113, 127, 150]. For example fault injection attacks performed on cryptographic algorithms (e.g. RSA [35], AES [123], PRESENT [144]) where the fault is introduced to reveal information that helps in computing the secret key.

Disturbing the hardware component of an embedded system can have an effect on the embedded software functionalities. This may impact the integrity and confidentiality of the system which will have consequences on the security, safety and privacy of the users behind.

Hardware disturbance is seen as a fault that is injected into the hardware which may lead to a modification of the program execution. Since the software is implemented in the hardware, any modification (fault) at the hardware level can lead to an exploitable error at the software level. In this thesis, fault injection is considered to be any perturbation at the hardware level which may modify normal execution of the embedded software.

Since embedded systems are essential and irreplaceable in many domains, it is important to ensure their dependability under fault injection. The concept of **dependable computing** [7] was used during the 1950's in the first generation of electronic computers. After discovering the first fault injections on hardware in 1954 [62], research was directed to have dependable computing and fault tolerance systems.

Later, laser was used as one of the first techniques to simulate the injection of faults and study the effect that fault injection may have at the software behaviour [61]. After fault injection was considered to be used as a technique to assess the robustness of systems. Since then new software and hardware fault injection approaches were proposed [47, 80, 81, 107, 108, 152]. The usage of fault injection techniques became an important step in the validation process of developed systems [5].

Fault injection is used as a technique to simulate the effect of hardware fault that can be induced due to the environment or by a malicious attacker. But at the same time fault injection is an approach to detect vulnerabilities created by injected faults in the hardware. Fault injection vulnerabilities are any vulnerability at the software level which is triggered by the hardware level fault injection. Detecting fault injection vulnerabilities is done using fault injection approach combined with other techniques, such as test or verification.

There are two main approaches to the detection of fault injection vulnerabilities: software-based and hardware-based approaches. Software-based approaches simulate fault injection on some aspect of the program and test/verify whether or not the injected fault yields a vulnerability in the program [31, 55, 107, 110]. Hardware-based approaches use direct experimentation on the hardware and program being executed inside, using hardware-based approaches vulnerabilities are observed by direct experimentation [10, 14, 114, 130].

Both approaches have advantages and disadvantages.

The advantages of software-based approaches are in cost, automation, and breadth. Software-based simulations do not require expensive or dedicated hardware and can be run on most computing devices easily [108]. Also with various software tools being developed and matured, compared to hardware experiments, it is easier to plug together a toolchain to do fault injection vulnerability detection [54, 55]. Such a toolchain can then be automated to detect fault injection vulnerabilities without direct oversight or intervention. Furthermore, simulations can cover a wide variety of fault models that represent different kinds of attacks and can therefore test a broad range of attacks with a single system. Combining all of the above allows for an easy automated process that can test a program for fault injection vulnerabilities against a wide variety of attack models, and with excellent coverage of potential attacks.

The disadvantages of software-based approaches are largely in their implementations or in the confidence in the feasibility of their results. Many software-based approaches have shown positive results, but are often limited by the tools and implementation details, with limitations in architecture, scope, etc. However, the biggest weakness is the lack of confidence in the feasibility of their results: software-based approaches have not been proven to map to genuine vulnerabilities in practice.

The advantages of hardware-based approaches are in the quality of the results. A fault injection that has been demonstrated in practice with hardware cannot be denied to be genuine.

The disadvantages of hardware-based approaches are the cost, automation, and breadth. To do hardware-based fault injection vulnerability detection requires specialised hardware and expertise to conduct the experiments. This is compounded when multiple kinds of attacks are to be considered; since different equipment is needed to perform different kinds of fault injection (e.g. EMP, laser, power interrupt). Further, hardware-based approaches tend to be difficult to automate, since the experiments must be done with care and supervision, and also the result can damage or interrupt the hardware in a manner that breaks the automation. Lastly, hardware-based approaches tend to have limited breadth of application; this is due to requiring many different pieces of hardware to test different architectures, attacks, etc. and also due to the time and cost to test large numbers of locations for fault injection vulnerability.

1.2 Motivating Example: Verify PIN Example

This section presents a motivating example that is used to illustrate how a single bit flip can change a normal behaviour of a program and create a vulnerability. The example is of a program that checks a PIN supplied by a user when authenticating to use a credit card. This example has been widely used in the literature [45, 117].

Consider the code in Figure 1.1 that checks the value of a candidate PIN entered by a user when authenticating to use a credit card. Prior to this code fragment the true PIN is assumed to be defined and initialised with the true PIN value. Similarly the candidate PIN `PINCandidate` is defined and initialised with a value input by the user. Further, both PINs are checked to be the same length and this length is defined to be their size `PINSize` (in the program the size of the PIN is set to 4, `PINSize = 4`).

The code fragment in Figure 1.1 starts by setting the variables `grantAccess` and `badValue` to `false`, and initialising the variable `i` to `0`. Which means that initially the access to the credit card functionalities is not given (`grantAccess = false`), and it is assumed that the two PINS values are equal (`badValue = false`).

Then, to check if the values of the two PINS are equal a `while` loop is used (line 4 in Figure 1.1). The loop iterate through the values of `PINCandidate` and `PINSize`, checks for each `i` iteration if the two values are equal (line 5 in Figure 1.1). If the iteration values of the two PINS are equal the code loops to the following one, if not `badValue` is set to `true` (line 6 in Figure 1.1). When all the values of the two PINS are compared, the code checks the value of the variable `badValue`. If `badValue == false` this means that the `PINCandidate` and `PINTrue` are the same, so the access can be granted (`grantAccess = true`), if not the access will remain denied (`grantAccess = false`).

```
bool grantAccess = false;                                    1
bool badValue = false;                                       2
int i = 0;                                                   3
while (i < PINSize) {                                        4
  if (PINCandidate[i] != PINTrue[i]) {                       5
    badValue = true;                                         6
  }                                                          7
  i++;                                                       8
}                                                            9
if (badValue == false) {                                     10
  grantAccess = true;                                        11
}                                                            12
```

FIGURE 1.1 – Motivating Example: VerifyPIN Source Code

Notice, that in line three in Figure 1.1, by changing a single bit, an attacker could change the value of `i` from `0` to `4`. (This succeeds since `0 = 0...0000` in binary, and changing the sixth bit from `0` to `1` yields `4 = 0...0100`.) Observe that this would bypass the loop since `i < PINSize` (i.e. `4 < 4`) would not hold, and therefore the checking of any digits of the candidate PIN. Thus, the example is vulnerable to this kind of 1-bit fault injection attack (as well as several other attacks that will be introduced later, see section 5.1).

The above paragraph describes a fault that can be injected into the executable binary that would allow the attacker to gain access even without the correct PIN. This research proposes and explains a process that can be used to detect such fault injection vulnerability.

1.3 Contributions

To detect fault injection vulnerabilities, this thesis combined three different domains: fault injection, formal methods, and binary lifting. This section presents the key contributions of this thesis. The first contribution is a general automated formal process that allows the detection of fault injection vulnerabilities in binaries. The second contribution is the exploration of correspondence between software and hardware approach in detecting fault injection vulnerabilities.

The rest of this section details the two contributions.

Automated Formal Process For Fault Injection Vulnerability detection in binaries

Software-based approach is cheaper, faster, and offers higher controllability over the injected fault. Despite software-based approach disadvantages, a lot of researchers are interested in exploring the simulation result using software-based approach [139, 141]. This explains the existence of a large number of software-based simulation tools [31, 46, 47, 65, 107, 118, 119, 128].

Similarly, formal methods and in particular model checking, are becoming increasingly popular for verifying the correct behaviour of systems. The majority of existing simulation tools use test oracles to check wether or not the fault injection created

vulnerability. Few combine formal methods with fault injection to detect fault injection vulnerabilities. **Associating formal methods to fault injection technique will allow formal proofs of the detected vulnerabilities**.

Despite all the existing software-based approaches to detect fault injection vulnerabilities in the literature [31, 46, 47, 65, 107, 118, 119, 128], there still no general approach developed for a broad use. The majority of existing works were realised in the scope of specific projects. No existing approach was proposed to be architecture independent. No existing approach uses model checking to detect vulnerabilities, related to the behaviour of the system, after simulating the fault injection.

The first contribution of this thesis is to improve the existing state of the art by proposing a general process that allows automated formal detection of fault injection vulnerabilities in binaries.

The motivation for an automated process is to facilitate the detection of fault injection vulnerabilities. An automated process will allow the evaluating of the robustness of codes during the whole development process. This will produce robust program (systems) against fault injection, with low cost. The correctness or robustness of systems at the last stages of development has a higher cost, being able to detect vulnerabilities at earlier stages saves a lot of money and time [7, 44].

Combining formal methods with fault injection detection ensures the rigour of the analysis and gives a guarantee to the detected vulnerabilities. Formally proving the presence of vulnerability is sufficient to claim the existence of software vulnerability.

Simulating fault injection at the binary level allows the simulation of faults which are representative of fault that can occur in physical fault injection. Contrary to simulating fault injection at the source code for example, simulation at the binary level allows the simulation of bit flips, this allows a fine granularity to the possible faults that can be injected. Working at the binary level gives a more realistic aspect to the detected vulnerabilities.

In addition to the proposed process, an implementation of this one that allows easy automation is given using existing open source tools and tools developed during this thesis.

Two tools were developed during this thesis. The SIMFI tool, a fault injection simulator tool, that allows the simulation of faults at the binary level based on a chosen fault model and generates the corresponding mutants. The ARML tool, an ARM to RML translator, that allows the translation of ARM assembly to Reactive Modelling Language (RML) which allows the formal verification of binary files. The two developed tools contributed in advancing the state of the art of available tools and pushing the bounds of some existing limitations.

A case study demonstrates the efficacy of using the process on cryptographic algorithms (PRESENT and SPECK), used widely in embedded systems. The conducted experiments detected known vulnerabilities on the two cryptographic algorithms, but also pointed out new ones that can be severe to the security of the system.

Combining Software-based and Hardware-based fault injection approaches

Combining fault injection and formal methods using software-based approach is not sufficient to claim that the vulnerability is real. Hardware experiments are needed to validate the existence of vulnerability in reality. Hardware experiments compared

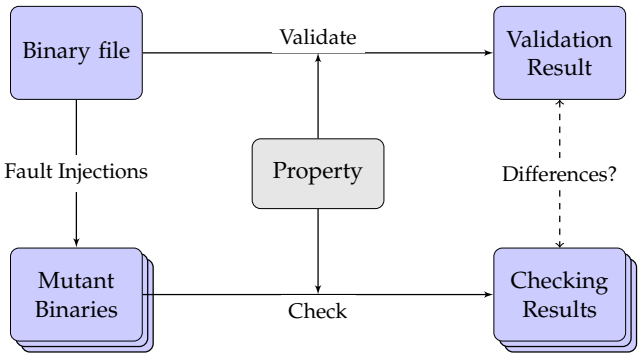

FIGURE 1.2 – FIVD process Figure

to simulation experiments are expensive, time consuming and requires a lot of expertise.

The second contribution of this thesis is proposing an approach that combines the software and hardware based approach to detect fault injection vulnerabilities in an efficient way. To our knowledge, no research has been done trying to bridge the software and hardware based approach to detect fault injection vulnerabilities the way it is done in this thesis.

Using both the software and hardware based approaches showed that:

— Software-based approaches detect genuine fault injection vulnerabilities.

— Software-based approaches yield false-positive results.

— Software-based approaches did *not* yield false-negative results.

— Hardware-based EMP approaches do *not* have a simple fault model.

— Combining software and hardware based approaches yields a vastly more efficient method to detect genuine fault injection vulnerabilities.

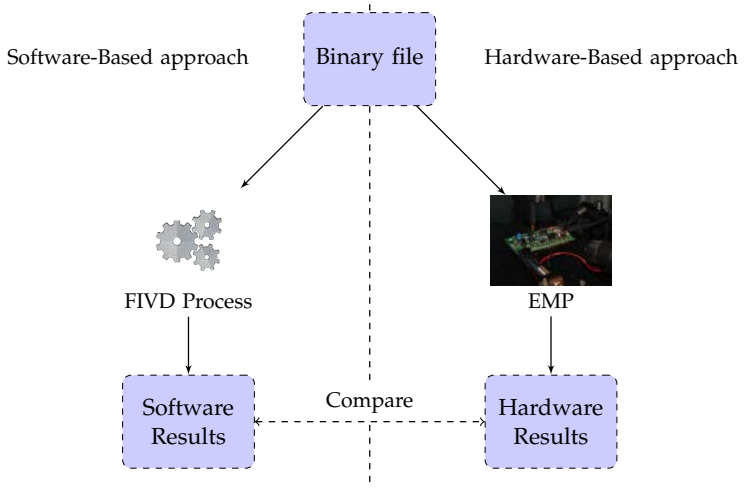

FIGURE 1.3 – Bridging Software and Hardware based approches

1.4 Publications

This section presents the list of papers published during this thesis. For each paper an abstract is given. The list of papers is classified in two categories: papers that presents the same results as in this thesis and thus related to the work presented

here, and a paper that represents collaboration that makes use of some contribution of this thesis but which are not related to the work present in this thesis. Note that the papers are not presented in the order of publication.

Papers related to this thesis

In [69] a formal approach that detects fault injection vulnerability was presented. The result presented in this paper was a proof of concept. It explores a first try to combine existing tools to implement the automated formal Fault Injection Vulnerability Detection (FIVD) process. The proposed implementation of the FIVD process was tested on a small example (the verify PIN example).

[55] presents the automated formal process for fault injection vulnerability detection. An implementation of the process is presented using existing tools. The feasibility of the process and its implementation is demonstrated by detecting vulnerabilities in the PRESENT cryptographic algorithm binary. The content of this paper is presented in Chapter 4 in this thesis.

[54] presents an extension of the automated formal process for fault injection vulnerability detection presented in [55]. In [54] the FIVD process was improved to make it scalable to real-world implementation. The scalability is demonstrated by detecting vulnerabilities in different cryptographic implementations (PRESENT and SPECK). Chapter 5 presents in more details the content of this paper.

[53] presents a new methodology of bridging the software-based and hardware-based approach for fault injection simulation. This paper presents an extension to the software-based process presented in [54, 55]. The implementation of the process in this paper proves that the process is architecture independent. The bridging methodology is applied of two different case studies. The experiment results shows that bridging the software-based and hardware-based approaches can save time, effort, and money, and get more accurate results. In this thesis Part II presents in details the combination of software-based and hardware-based approaches.

[69] gives an overview of the fault injection vulnerability detection techniques and tools. This paper can be considered as a survey that present the existing work and highlight the contributions of this thesis.

Paper not related to this thesis

An additional publication not completely related to the subject of this thesis were also published. This paper was a collaboration with another PhD student.

In [6] The objective was to verify the Dynamic Software Updating (DSU) system. DSU system consists in updating running programs without any downtime. This is needed specifically for critical applications that must run continuously. In this work model checking was used as a technique to verify if the system satisfies a list of properties (Deadlock, Safety, Liveness properties).

1.5 Organisation of the Thesis

This thesis is organised in two parts. Part I focuses on the software-based fault injection approach. Part II explores the potential of bridging the software-based and hardware-based fault injection approaches.

Prior to these areas (parts) is the background in Chapter 2. Chapter 2 introduces mains concepts needed to the understanding of this thesis, concept like fault injection, model checking, and binary translation.

Part I consists of 4 chapters. Chapter 3 introduces Part I of this thesis. Related works are presented to give an overview of existing works and how it is compared to the work presented in Part I. Cryptographic algorithms (PRESENT and SPECK) are presented in details in this chapter.

Chapter 4 presents the automated formal process for fault injection vulnerability detection FIVD and its extension adapted for larger programs. This chapter presents an implementation of the FIVD process and its extension using existing and developed tools.

Chapter 5 presents the experimental results of the FIVD process. First this chapter shows the feasibility of the FIVD process and its implementation on the Verify Pin motivating example. Then the scalability of the process and its implementation is demonstrated on the two cryptographic algorithms presented in chapter 3.

Chapter 6 gives details on the experimental results and implementation for the PRESENT Algorithm, it is complementary to Chapter 5.

Chapter 7 concludes Part I of this thesis and discusses some of the limitations of the FIVD process and the tools used in the implementation.

Part II consists of 4 chapters. Chapter 8 introduces Part I of this thesis. Related works that explore the combination of the software-based and hardware-based approaches are presented, to give an overview of existing works and how they compare to the work presented in Part II. Two case studies (Control Flow Hijacking and Backdoor) used in the experiments are shown as well.

Chapter 9 details the software-based and hardware-based processes used to conduct the experiments, and their implementation respectively. The software-based process used here is the FIVD process presented in the previous part with some improvement. The software-based process implementation uses ARML tool that was developed during this thesis. The hardware-based process is based on general process. The hardware-based process implementation uses the electromagnetic pulse to inject physical faults.

Chapter 10 presents the experimental results of applying the software-based and hardware-based approaches on the case studies. For each case study the results of the two approaches are shown separately and a comparaison of the two is made after.

Chapter 11 gives details on the experimental results and implementation for the control flow hijacking case study, it is complementary to Chapter 10.

Chapter 12 concludes Part II of this thesis and discusses some of the limitations of combining the two approaches (the software-based and hardware-based).

Chapter 13 provides a conclusion for both parts and a conclusion for a thesis as whole. It also presents a list of potential future work.

Chapter 2

Background

This chapter recalls three main key concepts needed in the understanding of this thesis. First, fault injection is presented in detail from the different type of faults to the existing fault injection approaches used to inject faults. Next, formal verification is introduced through different existing techniques, the focus is then on model checking techniques, since it is used in this thesis. Finally, working on binaries is presented through two perspectives, model checking binary and translating binary to an intermediate representation.

Sommaire

2.1 Fault Injection

Fault Types

Fault is defined in the ISO 10303-226 document [111] as: "an abnormal condition or defect at the component, equipment, or sub-system level which may lead to a failure".

A fault can be e.g., an accidental condition, a defect, or an unintentional short-circuit.

Accidental condition that occurs in the hardware can lead to a failure in performing a required function. An example is a fault that occurs due to deterioration or wear of hardware materials. This may concerns all products, it was shown that at a stage in its lifetime the failure rate of a product increases (bathtub curve) [147].

A defect in the construction process can cause a reproducible malfunction, which will occur consistently under the same conditions. This is usually a result of an error in the specification of the equipment and therefore affects all examples of that type. An example is the recent fundamental design flaw in Intel's processor chips [34],

that forced the redesign of the Linux and Windows kernels to bypass the chip-level security bug.

In power systems, fault can be an unintentional short-circuit between two adjacent interconnects or between energised conductor and ground. This kind of fault concerns all powered devices with an electronic circuit.

The focus in this thesis is not to a specific fault. The objective is to detect all kinds of faults.

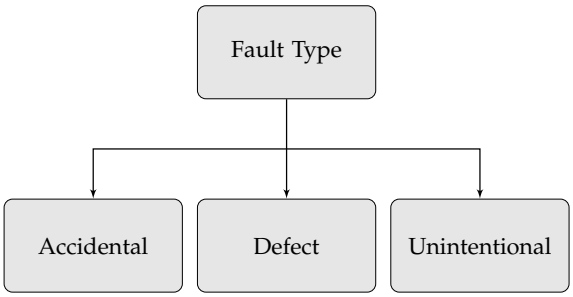

FIGURE 2.1 – Fault Types

Fault Causes

Fault can be caused due to problems occurring during the construction process of the hardware, or due to external factors [44, 72].

First, fault that occurs during the construction process. During the construction process numerous problems can occur either at the specification, implementation, or fabrication stages, which may lead to a fault.

Faults due to incorrect specification are called *specification faults*, this happen when the specification requirements ignore aspects of the environment where the system operates. It is possible that the systems with specification faults operates correctly most of the time, but they could be instances of incorrect performance. System-on-Chip (SoC) [115] are the common example for the specification faults, since the specification considered by the SoC vendors do not always contain all the details that SoC users need [124].

Faults due to incorrect implementation are called *design fault*, this happen when the system implementation does not appropriately implement the specification. These include poor component selection, logical mistakes, poor synchronisation, or bugs in the software. An example of design fault is the Ariane 5 rocket accident [87]. Ariane 5 is a European heavy-lift launch vehicle that exploded 37 seconds after lift-off on June 4,1996. The explosion was due to a software fault that resulted from converting a 64-bit floating point number to a 16-bit integer.

Faults due to fabrication defects or *component defects* were the primary reason for applying fault tolerance technique to early computing systems [44]. Fabrication defects are generally due to the chemical and physical reactions that the used material can have during the processing operation of fabrication [146]. But nowadays the development of hardware components has become more reliable and the percentage of fabrication defects faults was reduced [44].

Then faults that can be caused by external factors, such as environmental disturbances or human action, either accidental or deliberate.

Environmental disturbances arise from outside the physical system boundary. Such environments include aviation, military, space, etc. where atmospheric radiation, EMP, cosmic rays etc. may induce faults. An example is the Mars Pathfinder spacecraft where the mission was jeopardised by a concurrent software bug in the lander, the software bug being caused by a hardware glitch [48].

When the fault is due to an action that a person did, it can be accidental or deliberate. Accidental, when exposing your credit card to high temperature by accident, which may cause a disfunction in the card functionalities. Deliberate, when a person will expose the target system to electromagnetic radiation with the objective to bypass a security check for example.

In this thesis the focus is to detect the change of behaviour at the software level after a fault occurs. Note that this thesis does not focus on a specific fault cause.

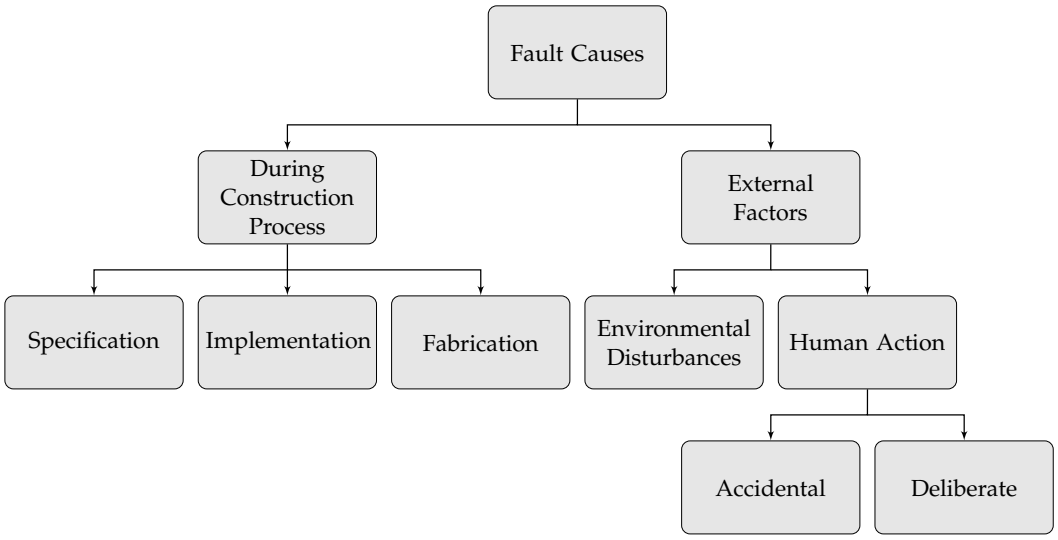

FIGURE 2.2 – Fault Causes

Academic Use

This section investigates fault and their use in academic research. The effect of faults on electronic systems was discovered accidentally in 1954, when radioactive elements present in packing materials caused bits to flip in chips [12]. In 1979 researchers were interested in studying the effect of cosmic rays on computer memory and see what impact they can have on the correct functionalities of chips [154]. Research showed that fault can have an impact on the correct behaviour of systems [5, 43, 72, 75, 154]. Fault injection was developed as a technique to test the fault tolerance of systems using faults. [1]

Injecting faults in systems has two main objectives: system validation and system evaluation [32]. In the first case, the objective is to test the fault tolerance mechanism designed to protect a system in presence of faults, and see if the implemented solutions are handling the faults they were designed to handle. In the second case, the goal is to evaluate system performance under fault, and see how it will behave after a fault injection.

1. In the literature four different terminologies exist for fault injection: fault insertion, fault injection, fault attack, fault injection attack. For the rest of this thesis the wording choice is fault injection.

Fault injection is a technique used to test and evaluate systems under fault [139]. In order to have reliable systems it is important to test them under fault and see their reaction. Fault injection was used first in the validation procedure of systems. Injecting faults physically in the hardware or simulating faults on the systems under tests yields three main benefits [32].

First, understanding the effects of real faults. Fault injection allows the reproduction of what may happen to the system in its real environment [61, 81]. Experiments are usually done in a controllable experimental environment, which allows an easiness of understanding the effect a real fault can produce.

Second, feedback for system correction or enhancement. Fault injection was first used to test the system dependability, and it showed its efficiency in evaluating systems and prove their correctness and dependability [5, 19, 32].

Third, forecast of expected system behaviour. Fault injection can also be used to predict the effect that faults will have on the system's behaviour. This can reveal the weaknesses in systems and help improving the system security [19, 59, 71, 117].

Faults lead a system to error and error can lead to system failure. "Failure is deviation of the component or system from its expected delivery, service or result that is due or expected" [139]. System failure can be classified to failure that will lead system to crash or disfunction, or failure that will lead system to perform unsecured operation and disclose secured informations. Failure can lead to system vulnerability.

A vulnerability is defined in the National Institute of Standards and Technology (NIST) Special Publication 800-30 [133] as a flaw or weakness in system security procedures, design, implementation, or internal controls that could be exercised (accidentally triggered or intentionally exploited) and result in a security breach or a violation of the system's security policy.

In this thesis, fault injection *vulnerability* is a fault injection that yields a change to the program execution that is useful from the perspective of an attacker. This is in contrast to other effects of fault injection that are not useful, such as simply crashing a program, causing an infinite loop, or changing a value that is subsequently overwritten. Observe that the definition of a vulnerability is not necessarily trivial or stable, the above example of a program crash may be a vulnerability if the attacker desires to achieve a denial of service attack.

To test/verify systems using fault injection, faults are injected either at the hardware level (logical or electrical faults) or at the software level (code or data corruption) [67]. Fault injection techniques can be divided into two approaches: software-based fault injection approach, and hardware-based fault injection approach.

2.1.1 Software-Based Fault Injection Approaches

This section presents the different classification of the software-based approaches, and investigates their advantages and disadvantages.

Software-based approaches consist of reproducing at software level the effect that would have been produced by injecting a fault at the hardware level.

From a first perspective, the software-based approach can be classified in two sub approaches: the software-implemented approach and the simulation-based approach.

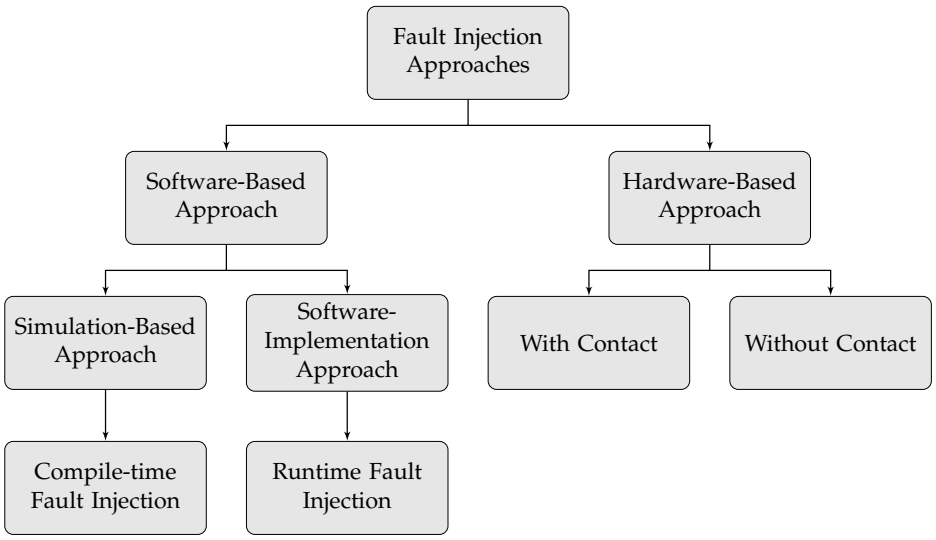

FIGURE 2.3 – Fault Injection Approaches

In this thesis, the simulation-based approach is used for the software-based approach.

The software-implemented approach consists in simulating the fault injection at software level while it is running on the target hardware. It consists of influencing the software application by altering its data or timing using an embedded fault injection simulator software while the target application is running on the target hardware. The software-implemented fault injection approach provides a cheap means to modify the software/hardware state of the target application while it is running on the target hardware without using any physical material. One major drawback of the software-implemented approach is the inability to inject faults to locations not accessible to software. Since only software is used to simulate the fault at the target application, the simulated faults are limited to locations where software can access.

In **the simulation-based** approach, the whole system behaviour is modelled and imitated using simulation. The simulation-based approach consists in taking the program and using software to build a model of its behaviour [80]. The faults may be injected into the program before or after the model is constructed, but the model is then tested for specific behaviours or properties and the results used to reason about the behaviour of the program. Simulation-based approach offers perfect controllability over the target system, and is becoming more popular as formal methods can be used on the model that allow for reasoning about all possible outcomes, and verifying when properties of the model may hold. Note that during the conceptual and design phase the simulation-based approach is useful for evaluating system's dependability [67].

From another perspective software fault injection attacks can also be classified into two kinds of fault injection attacks [67], *run time* and *compile time*.

Run time fault injection attacks are those that occur only while the code being attacked is being executed. Run time fault injection attack consists in injecting the fault while the program is executed. Compared to the classification presented before, run time fault injection will be possible only at the software-implemented approach. Compile time fault injection attacks are those that occur at any time starting form compilation of the code, and up until just prior to execution.

Compile time fault injection attack consists in injecting the fault before the program image is loaded and executed. Compared to the classification presented before, compile time fault injection corresponds to the simulation-based approach.

The advantages of software-based approaches are in cost, automation, and breadth. Software-based simulations do not require expensive or dedicated hardware and can be run on most computing devices easily [108]. Also with various software tools being developed and matured, limited expertise is needed to plug together a toolchain to do fault injection vulnerability detection. Such a toolchain can then be automated to detect fault injection vulnerabilities without direct oversight or intervention. Further, simulations can cover a wide variety of fault models that represent different kinds of attacks and can therefore test a broad range of attacks with a single system. Combining all of the above allows for an easy automated process that can test a program for fault injection vulnerabilities against a wide variety of attack models, and with excellent coverage of potential attacks.

The disadvantages of software-based approaches are largely in their implementations or in the veracity of their results. Many software-based approaches have shown positive results, but are often limited by the tools and implementation details, with limitations in architecture, scope, etc. However, the biggest weakness is the lack of veracity of the results: **software-based approaches have not been proven to map to actual vulnerabilities in practice**.

2.1.2 Hardware-Based Fault Injection Approaches

This section presents the different types in the literature of hardware-based fault injection, and investigates the advantages and disadvantages of the hardware-based approach.

Hardware-based approaches consists of disturbing the hardware at physical level, using hardware materiel (e.g EMP, Laser, Temperature, etc.). Hardware-based approaches are usually achieved by configuring the specific hardware to be experimented on and loading the program to be tested for vulnerabilities. A special device is then used to perform fault injection on the hardware during execution, e.g. EMP a chip, laser a transistor, overheat a chip. The result of the execution of the program is observed under this fault injection, with some particular outcomes considered to be "vulnerable" and thus a vulnerability is considered to have been achieved. One typical requirement for this approach is to have an idea of how a vulnerability is observable from program execution, since otherwise it is unclear whether the outcome of execution is a vulnerability or merely some normal or faulty behaviour.

Depending on the fault, hardware-based approaches fall into two categories[67]: hardware fault injection with contact and without contact.

In hardware fault injection with contact, the injector has direct physical contact with the target system. Examples are methods that use pin-level probes and sockets.

In hardware fault injection without contact, the injector has no direct physical contact with the target system. Examples are methods such as heavy-ion radiation and electromagnetic interference.

Various techniques exist to perform hardware fault injection either with contact or without contact [51, 67, 108, 152]. In this thesis, the hardware-based approach used is the electromagnetic pulse for the hardware experiments conducted in the later

chapters. Other technique are also presented in this section to give the reader other examples, and compare the chosen technique with the other existing ones.

Pin-level fault injection is believed to be the first technique used to inject fault [32]. Pin-level fault injection consists in changing the electrical signals at selected target device pins.

An other hardware fault injection attack, is when a fault in injected by tampering with the external clock signal of the target device. Two known ways of exploiting the clock signal for fault injection are: overclocking [60] and clock glitching [81]. Overclocking attacks consist in applying persistently a higher frequency clock signal then the normal clock frequency of the device. Clock glitching and contrary to overclocking consists in shortening the length of a single clock cycle. Both attacks causes the violation of the setup time constraints of the device and creates erroneous behaviour.

Fault injection through power supply is considered to be an inexpensive and sometime a natural way to introduce faults. This attack consists in altering the external power supply on the device, which can be done in two ways [108]: underfeeding or voltage glitch. This kind of attack can be used to skip the execution of an instruction.

One of the recent hardware fault injection method is the use of laser beams. Laser usage in fault injection attacks was first used to simulate radiation induced faults [61]. It was shown [94] that there is correlation between the results obtained from cosmic radiations and laser. Laser is able to inject faults in a very accurate and precise way.

An other hardware fault injection attack is fault injection trough electromagnetic fields. In electromagnetic fault injection, the faults are induced on the target through a fault injection probe which is placed above the target. The fault injection probe is designed as an electromagnetic coil, which induces eddy currents inside the target after receiving a voltage pulse. The electromagnetic pulse fault injection attacks is known to be less precise compared to the laser, although precise attacks can be conducted (up to a level of a single bit) but detailed knowledge of the target chip (device) in order to identify the precise point of attack with the precise parameters to set on the electromagnetic pulse hardware.

Other hardware approaches exist in the literature but will not be presented in this section such as: fault injection through temperature, fault injection through focused Ion Beams, fault injection through light.

The advantages of hardware-based approaches are in the quality of the results. A fault injection that has been demonstrated in practice with hardware cannot be denied to be genuine.

The disadvantages of hardware-based approaches are the cost, automation, and breadth. To do hardware-based fault injection vulnerability detection requires specialised hardware and expertise to conduct the experiments. This is compounded when multiple kinds of attacks are to be considered; since different equipment is needed to perform different kinds of fault injection (e.g. EMP, laser, power interrupt). Further, hardware-based approaches tend to be difficult to automate, since the experiments must be done with care and oversight, and also the result can damage or interrupt the hardware in a manner that breaks the automation. Lastly, hardware-based approaches tend to have limited breadth of application; this is due to requiring many different pieces of hardware to test different architectures, attacks, etc. and also due to the time and cost to test large numbers of locations for fault injection vulnerability.

2.1.3 Fault Model

This section gives an overview of different types of fault models. Each fault model describes an attack that can be conducted through a hardware or software fault injection.

In fault detection process, faults are simulated based on a fault model. Fault models are used to specify the nature and scope of the induced modification. A fault model has two important parameters, location and impact. The location includes the spatial and temporal location of fault injection relating to the execution of the target program. The impact depends on the type and granularity of the technique used to inject the fault, the granularity can be at the level of bit, byte, or multiple bytes.

According to their granularity fault models can be classified into the following kinds [118].

Bit-wise models: in these fault models the fault injection will manipulate a single bit. One can distinguish five types of bit-wise fault model [118]: bit-set, bit-flip, bit-reset, stuck-at and random-value. In the scope of this thesis the fault model that represents the bit-wise models is the *bit flip* (FLP) fault model that flips the value of a single bit, either from 0 to 1 or from 1 to 0, this fault model is an example of a Bit-wise model.

Byte-wise models: in these fault models the fault injection will modify eight contiguous bits at a time (usually in the same byte from the program or hardware perspective, *not* spread across multiple bytes). One can distinguish three types of byte-wise fault model: byte-set, byte-reset or random-byte. In the scope of this thesis three fault models represents the byte-wise models: the *zero one byte*, the *non-operation*, and the *unconditional/conditional jump*. The *zero one byte* (Z1B) fault model that sets a single byte to zero (regardless of initial value), this fault model is an example of a Byte-wise model. The *non-operation* (NOP) fault model that sets a byte to a non-operation code for the chosen architecture, this is an example of a Byte-wise fault model (but can also be implemented as a Wider model by changing the value of the whole instruction word). The *unconditional jump* (JMP) and *conditional jump* (JBE) fault models that change the value of a single byte in the target of an unconditional or conditional jump instruction (respectively), these are examples of Byte-wise fault models.

Wider models: in these fault models the fault injection will manipulate an entire word (defined for the given architecture). For this fault model a sequence of 8 to 64 bits will be modified depending on the architecture, e.g. changing the value of an entire word at once. This will typically target the modification of an entire instruction or single word value. In the scope of this thesis the fault model that represents the wider models is the *zero one word* (Z1W) fault model that sets a whole word to have the value zero (regardless of prior value).

2.2 Formal Verification

Formal verification in the context of both software and hardware systems is the act of proving or disproving the correctness of a system with respect to certain formal specification or property.

Formal verification techniques can be classified in three different techniques: manual, semi-manual, and automatic techniques.

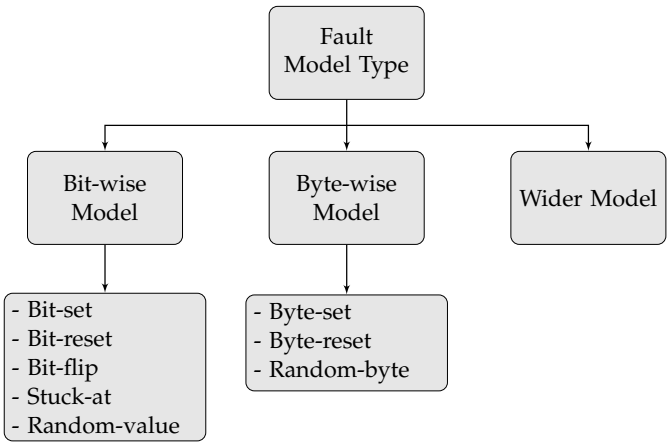

FIGURE 2.4 – Fault Model Types

Manual techniques are human directed proof. They are in general handwritten proofs in the style of mathematical proofs using natural languages.

Semi-manual techniques are when theorem proving is used to prove or disprove the correctness of a system. This technique is mainly based on deductive inference.

Automatic techniques are based on algorithm which are implemented in tools that take as input the model of the system and the property to verify, and decide if the model satisfies the property or not in return. In this thesis the focus is on automatic techniques, specially model checking

Formal verification techniques were applied in the last 20 years in variety of industriel domains [33, 148], particularly to verify the correctness of safety-critical systems [58], systems that can cause death, injury, or big financial loses (e.g., medical, automotive, and aerospace).

2.2.1 Model Checking

Model checking (MC) [9] is a formal verification technique used to verify if a given model satisfies specified properties. MC has the advantage that all possible states of the model are considered, and so is guaranteed to be able to answer whether or not a given property holds for a given model. Hence MC is an interesting technique for detecting the change of behaviour.

MC has become a standard method of analysing complex systems in many application domains. MC has demonstrated its efficiency in verifying systems [50, 149], although MC still has some limitations. Large or complex programs can have extremely large models that MC may fail to check in reasonable time [9], since MC is exploring every possible state of the model.

MC can fall into four classes: explicit state MC; Symbolic MC; Bounded MC; Constraint satisfaction MC.

Software model checking is the algorithmic analysis of programs to prove properties of their executions. Two approaches exist for software model checking [92]. The first approach model checks the actual software implementation by instrumenting either a simulator or the virtual machine for target architecture. The second approach directly instruments the machine code of the program and runs an analysis on the native hardware.

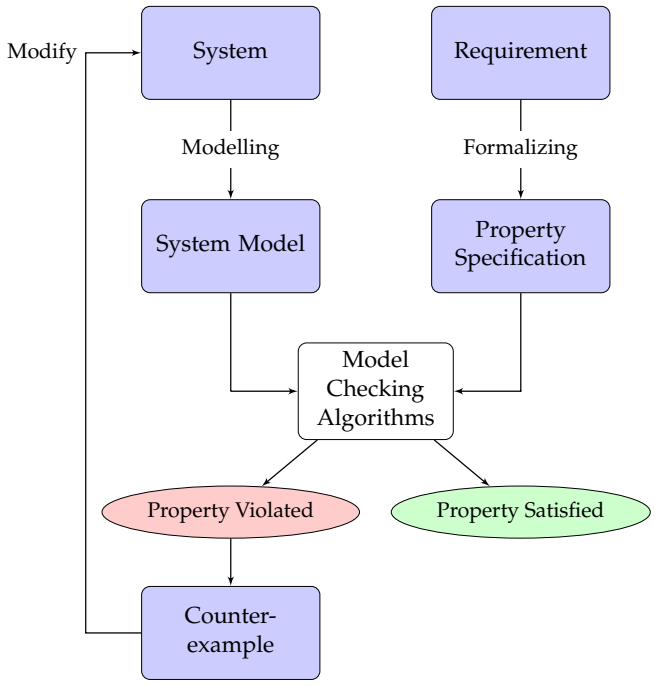

FIGURE 2.5 – Model checking diagram

Bounded Model Checking

Bounded model checking (BMC) is a refinement of model checking that alleviates some of the issues with possibly infinite complexity by bounding the checking [22]. The key idea in bounded model checking is to put a bound on parts of the model that could be infinite (or at least extremely large). For example, checking a program with a loop, going through hundreds or millions of iterations could be very costly for model checking. However, bounded model checking of such an example could limit the number of times to iterate through a loop. Thus, bounded model checking allows limits to be placed upon such potentially unbounded aspects of model checking.

Statistical Model Checking

Due to the limitations of using MC and BMC on large and complex programs, *Statistical Model Checking* (SMC) is an alternative approach that can rapidly find approximate results [85].

SMC is seen as a trade-off between testing and formal verification. The core idea of SMC is to conduct some simulations of the system and verify whether they satisfy a given property. The results are then used together with algorithms from the statistical area in order to decide whether the system satisfies the property with some probability. Of course, in contrast with an exhaustive approach, a simulation-based solution does not guarantee a result with 100 confidence. However, it is possible to bound the probability of making an error. Simulation-based methods are known to be far less memory and time intensive than exhaustive ones, and are sometimes the only option. Over past years SMC has been used to 1. Assess the absence of errors in various areas from aeronautic to systems biology, 2. To measure cost average and energy consumption for complex applications such as nanosatellite and 3. Detect rare bugs in concurrent systems. The approach is now widely studied in academia, It is also used in research projects and endorsed by industry (IBM, THALES, EADS ...).

2.2.2 Properties

In MC, BMC, and SMC properties are used to define the correct or incorrect behaviour of the model. Here properties are used to define specific vulnerabilities that may be introduced by fault injection.

To perform model checking, the *properties* to be checked upon the model need to be specified. There are two main kinds of properties that can be checked: *safety*, and *liveness* [9]. Safety properties are used to express that certain propositions hold when they are encountered. Liveness properties express that propositions hold over some temporal dimension. This thesis only considers safety properties since these are clearer, more intuitive to represent, and sufficient to illustrate the feasibility of the process. Liveness properties can also be checked in a similar manner, although this is not presented in this thesis.

Safety properties can be expressed by simple propositions that can be annotated into the code of the program being considered. Recalling the motivating example (in Figure 1.1 on page 8), a naive safety property could be expressed by an assert statement such as

```
__llbmc_assert(i == 4);
```

that is inserted between lines 9 and 10 in Figure 1.1 on page 8. This property would be checked by the model checker to ensure that the variable `i` has the value `4` at this point in the model.

More generally such asserts support properties defined with boolean propositions. Here we exploit properties supporting: negation denoted `!`, equality denoted `==`, inequality denoted `!=`, conjunction denoted `&&`, and disjunction denoted `||`. For example, the following property

```
__llbmc_assert(    !(PINCandidate != PINTrue) || grantAccess == false);
```

combines negation with inequality, disjunction, and equality. The semantics are that when the two PINs are not equal, then access is not granted.

In the second part of this thesis, the properties are specified using *Bounded Linear Temporal Logic* (B-LTL). B-LTL is chosen here for being able to represent the key concepts required and as it is used in the first part that uses the same foundations as exploited in the paragraph above. The properties in the second part of this thesis are mostly specified using simple (in)equality relations, however the temporal and bounding operations can be exploited to account for infinite loops induced by fault injection.

2.3 Working on Binaries

Working at the binary level is considered to be an efficient way to test, analyse, and verify the correctness and/or robustness of the developed systems in a variety of domains.

Binary code represent the last form of the program at the development cycle. Hence, all the errors (that affect the correctness of the system) and flaws (that represent a threat to the robustness of the system) which were introduced during the development process can be found.

Tool Name	Ref	Input	Architecture	Properties
JPF	[143]	Java bytcode	JVM	safety
StEAM	[91]	C++	-	safety
Arcade	[21]	compiled binary	ATmega16, ATmega128, R8C/23	safety
mc-square	[126]	C	ATmega16	safety

TABLE 2.1 – List of Some Existing Binary Model Checkers

The ideal solution is to detect vulnerability at the binary, since it is the most representative at the hardware level. Applying the fault injection vulnerability detection process at the binary will allow the detection of vulnerabilities which could not be found at the source code level for example.

In the scope of this thesis, one of the main objective is to be able to use formal methods (model checking) directly at the binary level to detect fault injection vulnerabilities. This introduced many constrains. The rest of this section presents a literature review of model checking binary. A list of advantages, disadvantages, and limitations is given. An explanation of why it is needed to go through an intermediate representation of the binary is given.

2.3.1 Binary Model Checking

Model checking is recognised as promising technique for the verification of systems in a variety of domains [9, 50, 78, 116]. The majority of existing model checker tools operates at a specific modelling language [66, 86], or on source code [13, 93]. But recently researchers were also interested in applying model checking to the binary level for various reason: first, when the source code is not available. Second, to be able to detect errors that might be introduced in the compiling process. Third, to detect malicious code inside executables. Finally, related to this thesis, to be close to the hardware representation.

Model checking binaries comes with many constrains. Model checking low level language adds hardware specification, which means that it needs to be adapted for every new hardware architecture. Beside the state space tends to be bigger than when model checking higher level language as more details are involved.

An example of model checking assembly is in [149], here the authors proposed the following method: Generating models including block cycles; Abstract and refinement method of bit level; Generating exact models by dynamic program analysis; Verifying model by model checking while generating the model by dynamic program analysis.

In the following paragraph a list of model checkers that handle binaries will be given (see Figure 2.1), highlighting their main limitations. In the scope of this thesis, an ideal model checking tool is a tool that takes binary file as input, that supports multiple architecture instruction sets (specially ARM or X86), that allows the verification of liveness and safety properties.

Java PathFinder (JPF)[143] is a verification framework dedicated for Java programs, it takes as input executable Java bytecode programs. It allows the verification of general software safety properties such as exceptions, deadlock, and user defined assertions which are specified in the source code directly. The main limitation of JPF is that it's limited to the executable Java bytecode programs it does not check executable ARM or x86 binaries, and it does not support liveness properties.

State Exploring Assembly Model checker (StEAM)[91], takes the C++ source code as an input. StEAM performs model checking on the assembly level compiled from C++ source code. In case the user had only the assembly code it will not be possible to verify it, since it needs the source code as input.

Aachen Rigorous Code Analysis and Debugging Environment (Arcade)[21], is a framework for the verification and analysis of embedded software. It verifies binary code for microcontroller. As input, it takes compiled binary code. Arcade is limited to a restricted list of supported architecture (X86 and ARM architecture are not supported), it only verifies safety properties.

mc-square [116, 126] is a model checker for microcontroller-based embedded systems. mc-square model checks assembly code that is compiled from C source code files. It is considered to be an old version of Arcade. mc-square is limited to the ATmega16 architecture and supports only safety properties.

From all the presented above one can conclude that model checking binaries is possible but not feasible for real world programs. The majority of the tools presented above can not work directly on binary files, source code file is always needed, which can not be practical to verify binaries without access to their source code.

To overcome the restriction of model checking binaries, proposed solution in the literature is to go through an intermediate representation of the binary [77]. The idea is that instead of model checking the binary code directly, one would translate the binary to an intermediate representation then model check the intermediate representation code.

A large list of possible intermediate representation languages exists (ASM, QUEMU-IR, etc). One of the famous intermediate representation which offers a large number of tools in terms of choices is LLVM-IR. LLVM-IR is an abstract assembler language that is architecture independent. Here LLVM-IR is chosen as an example that considers the first part of this thesis. The rest of this section will be presenting a list of model checkers that handle the intermediate representation language LLVM-IR.

The first presented tool is Divine [13], which explicit-state model checker based on the LLVM toolchain. Divine is a tool for Linear Temporal Logic (LTL) model checking and reachability analysis of discrete distributed system. Divine does the verification of safety properties and also some liveness properties. But the verification of liveness properties are limited to a restricted list of input languages, using the LLVM-IR as an input language it will not be possible to specify liveness properties. An other drawback with the Divine tool are potential problems one might have with the different version of the LLVM-IR intermediate representation language. The LLVM-IR version used by Divine is a modified version of LLVM-IR, this can cause problems of incompatibilities with other tools used in the toolchain.

The second presented tool is the Low-Level Bounded Model checker (LLBMC) which is a bounded model checker for the intermediate representation language LLVM-IR. LLBMC is used in the process implementation presented in the first part of this thesis. LLBMC checks only safety properties. The properties are specified as assertion statement in the source code. LLBMC only does bounded model checking, and so will not explore every possible execution of a program if this is beyond the bounds imposed by the execution.

Other tools exists in the literature but will not be detailed in this section such as SymDivine[99], which is a model checker, built as well upon LLVM compiler infrastructure. It is a standalone frame for LTL verification of LLVM-IR programs.

Its name is maybe similar to Divine tool, but SymDivine is not built on the Divine model checker

2.3.2 Binary Translation

As discussed in section 2.3.1, it is difficult to model check binaries directly. Prior works have proposed a solution which is to go through an intermediate representation.

Binary translation tool translate a binary code into another target language. Two method of binary translation exists [36]: static and dynamic.

Static method, aims at doing the translation without having to run the code first. This method does not guarantee a correct translation since not all code fragments can be discovered by the translation, some parts of the executable may be reachable only through indirect branches, whose value is known only at runtime. A limitation of a static translator is its inability to accurately account for all the code because some code paths cannot be predicted statically [70].

The dynamic method, translated code that is discovered through the execution.The dynamic binary translation looks at short sequence of code. The dynamic method is known to be better when having to translate machine code from one architecture to another. This will avoid doing work on code that never executes. Dynamic translation is slower than static, not always 100% accurate, some code path may be dependent on a specific set of input parameters. It suffers in term of performance.

In this thesis, the choice was made to adopt the static method for translating binary.

Developing a binary translator can be done in two ways [63]: manually or automatically. The manual method, consists in manually developing the code to translate each assembly instruction to its corresponding IR. The manual method is widely used [18, 88, 102], such a manual approach can be very laborious and requires a lot of time and effort, but it provides correct and reliable results. Manual method as shown in Figure 2.6 consists in: first, disassembling the binary code to the corresponding assembly code. Second, generating the corresponding Control Flow Graph (CFG) of the assembly code. Finally, developing manually the corresponding IR translation for each assembly instruction.

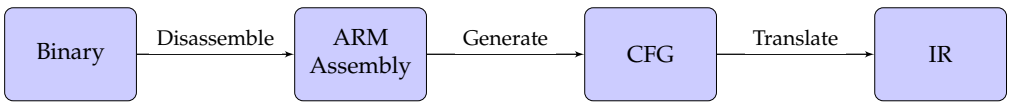

FIGURE 2.6 – Manual Binary Translation

The automatic method is a new approach that was introduced by Hasabnis et al. [63, 64]. The automatic approach consistes in using the knowledge already contained in compilers such as GCC [57] and LLVM [84]. Compared to the manual method, the automatic method is faster, does not requires coding efforts, but the correctness of the translation is not provided.

Automatic methods as shown in Figure 2.7 first train their learning algorithms by generation assembly code from an IR. In this step the learning approach memorise the exact translations observed in the training data. Then it uses the learning algorithms to translate from the assembly code to the IR.

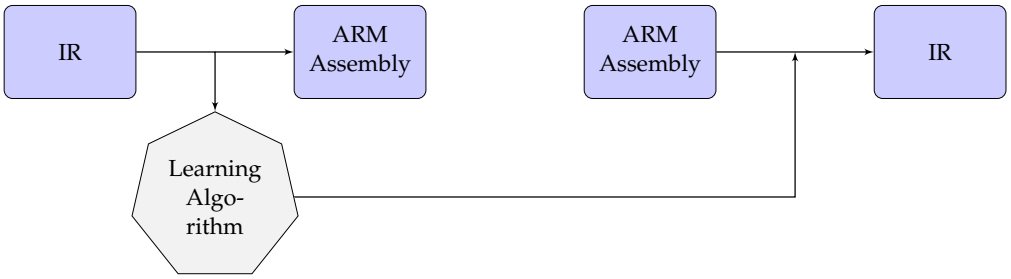

FIGURE 2.7 – Automatic Binary Translation

Part I

An Automated Formal Process For Detecting Fault injection Vulnerabilities in Binaries

Chapter 3

Overview of Part I

This chapter introduces the first part of this thesis. First, it gives a general introduction to the first part of the thesis which focuses on the fault injection software-based approach. Next, it reviews the literature of existing works that propose fault injection software-based approaches. Finally, it presents the two cryptographic algorithms used later in the experiments.

Sommaire

3.1 Introduction

Fault injection has been increasingly used to test the robustness of software systems. Many systems are particularly vulnerable to fault injection attacks due to operating in hostile environments, i.e. environments where an attacker may be able to perform physical attacks on the system hardware. Many such attacks have been demonstrated on various systems, showing that different kinds of faults can be injected into various devices [12, 59, 141]. Attacks can also be achieved through software alone and do not require attacking the hardware directly. A recent example of this is rowhammer [76] that has been exploited in various attacks [127, 150].

The wide variety of fault injection attacks and possible impacts upon a system make it impossible to prevent software from failing under any possible attack [141]. Thus, recent work has approached the problem of fault injection by limiting the scope of attacks, or limiting the kinds of vulnerabilities analysed [15, 35, 96, 97], often requiring specialised equipment.

The first part of this thesis explores a new software approach to broadly detect fault injection vulnerabilities using formal methods at the binary level. As a contribution an automated process for detecting vulnerabilities in binaries using model checking named FIVD.

The FIVD process is achieved by simulating fault injection attacks upon the executable binary for the given software, and then using model checking on the resulting executable binaries to determine whether or not the simulated fault injection attack violates properties which the software should maintain.

The FIVD process begins with the *executable binary* that represents the program to be considered. The *validation* of the executable binary involves checking various *properties* using model checking to ensure the executable binary meets its specification. Fault injection attacks are then simulated on the executable binary, producing *mutant binaries*. The properties are then model checked on the mutant binaries that pass pre-analysis. A difference in the result between validating and checking the properties indicates a vulnerability to the fault injection attack that was simulated.

This process provides a general approach that can support detecting a wide variety of fault injection vulnerabilities in binaries by varying the fault model of the fault injection while being scalable to real-world implementations. The strengths of this approach includes the following. By operating directly upon the binary, fault injection vulnerabilities that cannot be detected in source languages or intermediate representations can be detected [55, 119]. Formal methods, in this thesis model checking, ensure the rigour of the analysis and so ensure that fault injection vulnerabilities that are detected are real and not false positives. An automated process can be easily iterated over various fault injection models and approaches, and thus allows broad, or even complete, coverage of possible fault injection attacks. Combining automation, broad coverage, and formal methods, allows the process to make strong guarantees about the vulnerability of a system that has been analysed. The process design, and extension with pre-analysis, allow for easy parallelism and thus scalability of the process in practice.

To demonstrate the efficacy of this process, this part includes a case study of applying the process with various fault models to two different cryptographic implementations: the PRESENT lightweight encryption algorithm [25, 79], and the recently introduced lightweight encryption algorithm SPECK [17].

The results found 82 vulnerabilities, with 9 in each of PRESENT and SPECK allowing an attacker to bypass the encryption entirely. A further 64 cryptanalytical attack vulnerabilities were found in PRESENT.

This part presents the first version of the FIVD process (section 4.1), and improves it. Note that the two processes differ only in the way properties are specified. This difference is related to the implementation of the process and not the process itself. The choice in this thesis to present the two processes (two implementations) is to take the reader into the evolution of this work. First, it presents the first process, point the problems and limitations. Then, it shows how the problems were solved, how the imitations were overcome.

3.2 State of the Art

This section recalls existing work that uses software-based approach in order to evaluate the robustness of systems against fault injection.

In the literature, works that use the software-based fault injection approach to detect fault injection vulnerabilities exist. In order to be able to compare the proposed FIVD process with some of these works, a list of goals to meet is specified:

1. First, the proposed approach needs to be automated.
2. Second, the proposed approach needs to operates directly at the binary level.
3. Third, the proposed approach needs to support a large variety of fault models.
4. Fourth, the proposed approach needs to be architecture independent.

5. Fifth, the proposed approach needs to provide reliable results.

Perhaps the closest to the FIVD process presented here is the Symbolic Program Level Fault Injection and Error Detection Framework (SymPLFIED) [107] a program-level framework to identify potential vulnerabilities in a software. The vulnerabilities are detected by combining symbolic execution and model checking techniques. However, the SymPLFIED framework is limited as SymPLFIED only supports the MIPS architecture [112]. One of the proposed future work in [151] was building front-end to support X86 architecture, but to the best of the author's knowledge, no further work has been published on supporting new architectures in SymPLFIED.

A fault model inference focused approach is taken by Dureuil et al. [47]. They fix a hardware model and then test various fault injection attacks based upon this hardware model. Fault detection is limited to EEPROM faults on the ARMv7-M architecture. The fault model is then inferred from the parameters of the attack and the embedded program. The faults are simulated upon the assembly code and the results checked with predefined oracles on the embedded program. Although the approach uses neither formal methods nor general fault models, the choice of fault model and derivation of this provides some interesting guidance for selecting fault models and future work.

An entirely low level approach is taken in [97] which uses model checking to formally prove the correctness of their proposed software countermeasures schemes against fault injection attacks. The approach has some similarities to the approach presented in this thesis: using model checking while focusing on low level representations. However, the focus is on a very specific and limited fault injection model that causes instruction skips and ignores other kinds of attacks. Further, the model checking verifies only limited fragments of the assembly code, and not the program as a whole.

In [145] the authors present a fault injection simulation tool and they compare it with existing tools (Xception [31] or FERRARI [74]) which inject the fault at the binary level. The big advantage is that it supports multiple architecture. This paper presents FITgrind a fault injection tool that uses dynamic binary instrumentation provided by Valgrind [101]. The authors claims that the tool is architecture independent but since it is based on Valgrind, it is limited to the architectures supported by Valgrind. The tool does not simulate the fault directly on the binary but on the Valgrind intermediate representation which limits the fault model used. For example, it is not possible to simulate a bit-flip using this tool since Valgrind automatically rejects invalid code. To see if the fault injection had an effect on the program or no, the output of a fault injection run and the golden run are compared byte wise, any difference in considered as an error. This can not detect if the modification will create a vulnerable behaviour or will have no effect on the program behaviour.

An other work which has similarities with the proposed approach in this thesis is [110], where Lazart is presented, a tool that can simulate a variety of fault injection attacks and detect vulnerabilities using formal methods. The Lazart process begins with the source code which is compiled to LLVM-IR. The simulated fault is created by modifying the control flow of the LLVM-IR. Symbolic execution is then used to detect differences in the control flow, and thus detect vulnerabilities. Although this high level approach is similar to that of this thesis, Lazart is unable to reason about or detect fault injection attacks that operate on binaries rather than the LLVM-IR. Further, the choice of symbolic execution does not account for concrete values, and so is less complete than model checking [106, 134].

[140] has also the objective of assessing the robustness of systems using fault injection technique. Here the authors used Property-based testing and fault injection to test the robustness of the systems. They simulate the injection of the fault on the source code, then specify properties that the source code need to respect, after checking it using property based testing. The fact that the fault are simulated at the source code will not be representative of real faults that can be created in real environment. Beside property based testing does not formally prove that the software fulfils its specification, contrary to formal techniques like model checking.

In [118] the authors propose a generic fault injection simulation tool. The proposed tool embeds the injection mechanism into the smart card source code. It only supports data and control flow fault model. The EFS tool used in this paper performs the implemented-based approach for fault injection which differs of the simulation approach used in the FIVD process. It inject faults directly on binary but can not be architecture independent and is limited to specific fault models.

One of the first uses of formal methods to analyse fault injection vulnerabilities was to verify a counter measure in the implementation of CRT-RSA by analysing the C source code [35]. The authors show that by adding ANSI/ISO C Specification Language (ACSL) [16] properties to the CRT-RSA pseudocode, they could verify that the Vigilant's CRT-RSA countermeasure sufficiently protects against fault injection attacks. The lack of fault injection and analysis on the binary limits the attacks that can be detected to those that have a representation in the source code. Further, the analysis was only for a single countermeasure to prove it worked, rather than to consider a variety of fault injection attacks and models as is the goal of the process presented in this thesis.

In [68] the authors propose a technique along with a prototype fault injection tool to facilitate robustness evaluation of software. The proposed technique in this paper is limited to the AUTomotive Open System ARchitecture (AUTOSAR) based systems and it was not shown that it can be applied to other systems. As described [68] the simulation of the fault is not done by injection faults into the program but by giving erroneous input. This technique is considered to be fuzzing and not fault injection.

3.3 Case Studies: Cryptographic Algorithms

For embedded systems, it is important to ensure the security of stored and manipulated data. Cryptographic algorithms are considered to be resistant to direct attacks on target implementation [11], such as exhaustive key search and cryptanalysis. Because of the constrained nature of embedded systems, it is necessary to use algorithms that do not require high computational power. Lightweight ciphers are therefore ideal candidates for this purpose.

For this thesis, the choice was done on two lightweight block ciphers: PRESENT [25, 79], and SPECK [17]. Both cryptographic algorithms are designed for use on low power and CPU constrained devices, and so suitable for embedded systems. The objective is to evaluate the robustness of the chosen block ciphers against fault injection.

Fault injection is an efficient way to attack cryptographic algorithms in order to retrieve information about secret key. Fault injection exploits a possibility to change the intermediate values in the algorithm execution so that it can reduce the key search space or even reveal the key.

This section gives a brief overview of the encryption algorithms considered in the experiments (PRESENT and SPECK). Both implementations are open-source and consider a standard setting (e.g. without additional countermeasures). Some known cryptanalytic attacks on both algorithms will be also presented.

PRESENT

PRESENT is a lightweight block cipher based on Substitution-Permutation Network (SPN) [25, 79]. PRESENT was proposed by Bogdanov et al. at CHES 2007, and was standardised by IEEE.

The PRESENT algorithm consists of 31 rounds of a SPN with block size of 64 bits. The canonical implementation [1] supports key lengths of 80 or 128 bits. The core encryption algorithm is the same for both 80 and 128 bit keys.

The PRESENT algorithm round function consists of three stages (see Figure 3.1). **addRoundKey** which consists of XORing the 64 bits output of the last round function with the subkey; **sBoxlayer** which consists of a nonlinear substitution layer with sixteen 4-bit Sbox; **pLayer** which consists in permuting bits based on the permutation function.

The version of PRESENT analysed here is the canonical version in C for 32 bit architectures (size optimised, 80 bit key). The C implementation consists of one main loop, that will iterate the first 30 rounds. The 31st round will be performed after the end of the loop. Each iteration of the loop consists in performing first the three stages (**addRoundKey**, **sBoxlayer**, and **pLayer**), then the key scheduling that will compute the subkey used in the following iteration of the loop. The last 31st round consists only of the addRoundKey stage, that will XOR the output of the last round function with the subkey

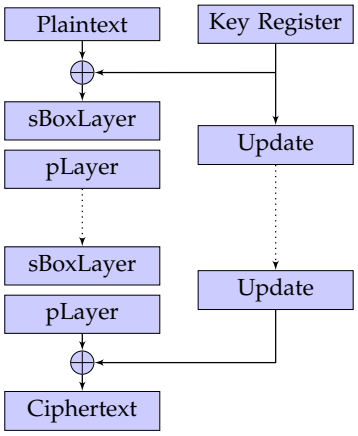

FIGURE 3.1 – PRESENT Algorithm Figure

SPECK

SPECK is a lightweight block cipher based on Feistel network [17]. SPECK was proposed by the National Security Agency (NSA) in 2013.

The SPECK algorithm is highly software-oriented as it relies only on additions, word rotations, and XOR operations. As illustrated in Figure 3.2 each round consists of:

1. Available at http://www.lightweightcrypto.org/implementations.php

— 2 rotations; — addition of the right word to the left word; — XORing the key into the left word; — XORing the left word to the right word. SPECK supports a variety of block and key sizes with respective number of rounds.

The canonical version [2] analysed here is implemented in C, has a key length 128 bits, block size 64, and 21 rounds. The C implementation consists of two main loops. The first loop consists of 20 iterations, that computes the subkey for each round. The second loop performs the encryption by following the steps shown in Figure 3.2 using the subkeys generated by the first loop.

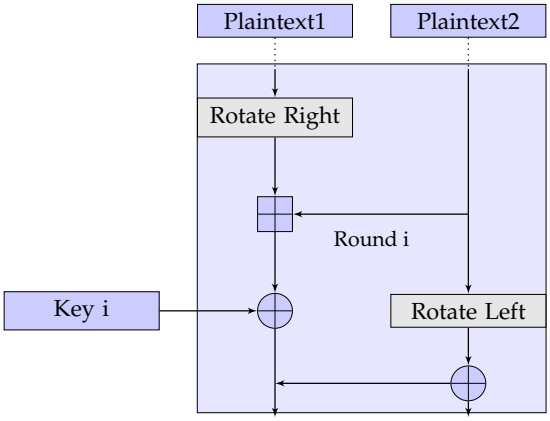

FIGURE 3.2 – SPECK Algorithm Figure

Cryptanalytic Attacks

Injecting fault into cryptographic algorithm was first proposed in [26, 27]. Where Boneh proposed a new cryptanalytic attack which exploits computational errors to find cryptographic keys only for public key crypto systems such as RSA. One year after [23] proposed a practical attack called Differential Fault Analysis (DFA) and showed that is applicable to any cryptographic algorithm.

Until today DFA is still the most popular technique for attacking symmetric block ciphers. Combined with fault injection DFA offers an efficient way to attack cryptographic implementation and try to break them.

The principle of DFA is to induce faults into cryptographic implementation to reveal their internal status. The attacker gets information about the secret key by examining the differences between cipher text resulting from a correct encryption and cipher text of the same plaintext and key resulting from a faulty encryption.

Many DFA were shown in the literature on PRESENT [8, 52, 144] and SPECK [1, 24, 42, 138]. For PRESENT it was shown in [8] that injecting a fault in a single bit at the beginning of the last round of the S-box layer allowed a Differential Fault attack (DFA). In [28] the authors were able to recover the secret key with two faulty encryptions and an exhaustive search of 216 remaining key bits. For SPECK, in [138] it was shown that by using a bit-flip fault model it is possible to recover the last round key. In [11] the authors describes a simple platform for the study of fault injection and analysis in the context of fault attacks block ciphers based on a Feistel structure.

2. Available at https://github.com/inmcm/Simon_Speck_Ciphers

More general approaches to detect fault resilience in block ciphers were also proposed in [29, 40, 73, 98, 121]. In [98] the authors showed fault attack against PRESENT, Simon, and others. In [29] the authors propose an automated approach for analysing cipher implementations in assembly. The authors implemented their approach on a tool called DATAC, they used it to attack the PRESENT algorithm and to find implementation-specific vulnerabilities. In [40] the authors present a fault injection attach against PRESENT. In [121] the authors propose a novel fault injection attack against AES using a new fault model. The fault injection attack is then combined with a side channel attack to gain the secret key. [73] here the authors present a fault injection attack on LED clock cipher which will reduce the number of key and then will make it possible to deduce the key by a brute force.

Chapter 4

Process, Implementation and Methodology

This chapter presents the automated formal process for detecting fault injection vulnerabilities in binaries (FIVD). The FIVD process is one of the main contributions of this thesis. This chapter first presents in details each step of the process and argues the choices that were made. Then, it details the first implementation attempt of the FIVD process using existing tools. Next, it presents the methodology followed to run the experiments. Additionally, this chapter also highlights some minor improvement done to the FIVD process and its implementation to make it scalable for larger programs.

Sommaire

4.1 Fault Injection Vulnerability Detection *FIVD* Process

This section presents in details the FIVD process, an automated formal process for detecting fault injection vulnerabilities in binaries. The FIVD process presented in this section was conceptualised as a proof of concept to show the feasibility of the process.

An overview of the FIVD process is shown in Figure 4.1. The FIVD process is as follows.

Prior step to starting the FIVD, the properties to check on the binary must be defined, and then annotated in the source code. Property specification is an important step, since it is what defines the behaviour the process checks. Property specification is related to the target program. Property can not be general to all programs (systems). To specify the property to check, a user needs to have an idea of the target program, the user needs also to have a list of behaviours that the program (system) should or should not respect.

A property might specify a good or bad behaviour. When property specifies a good behaviour, it is in general a behaviour that the program needs to respect under fault injection. An example related to the motivating example (Figure 4.3) can be : verify that if `PINCandidate` is equal to `PINTrue`, then `grantAccess` is equal to `true`. In this case, if the program satisfies the property then fault injection does not create a

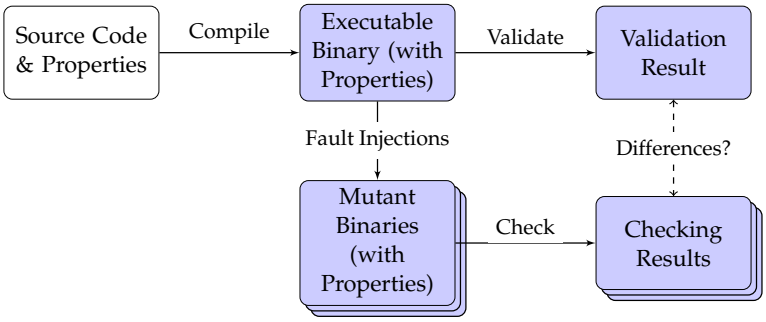

FIGURE 4.1 – FIVD Process Diagram

vulnerability in the target program. If the program violates the property then we conclude that fault injection created a vulnerability in the target program. When a property specifies a bad behaviour, it is in general a behaviour that the program should never have even under fault injection. An example related to the motivating example (Figure 4.3) can be : verify that if `PINCandidate` is not equal to `PINTrue`, then `grantAccess` is equal to `false`. In this case, if the program satisfies the property then fault injection created a vulnerability in the target program. If the program violates the property then we conclude that fault injection didn't create a vulnerability in the target program.

A preparation step for the process is then to compile the source code and properties to produce an executable binary in a manner that preserves the properties as annotations. The produced binary file is the process input file within properties to verify.

The first step of the process is the validation of specified properties to hold upon the executable binary before the simulation of fault injection. This ensures that the executable binary meets the specification of the properties. If there is some other error in the source code or compilation, this can be detected here and not be (incorrectly) attributed to fault injection vulnerability. During this step the executable binary within the properties is given to model checker tool to validate that the executable binary satisfies the checked properties.

The second step of the process consists of simulating the injection of faults on the executable binary to produce mutant binaries. During this step the objective is reproducing the effect of the fault injection upon the executable binary. This simulates the actual fault injection attacks and produces mutant executable binaries that represent the executable binary after the fault injection simulation.

The third step of the process is checking the properties upon the mutants binaries. All generated mutant binaries are checked using model checker to satisfy the specifies properties or not. This step is similar to the first step with the difference that instead of the executable binary, the mutant binaries are checked.

The fourth step of the process consists of comparing the validation and checking results. A difference between the validation of the executable binary and checking the mutant binary indicates a vulnerability to the simulated fault injection. To conclude that the fault injection had no effect on the behaviour of the executable binary, the result of validating the executable binary and checking the mutant binary needs to be similar. If the results is different then the simulated fault injection succeeded in creating a vulnerability in the program.

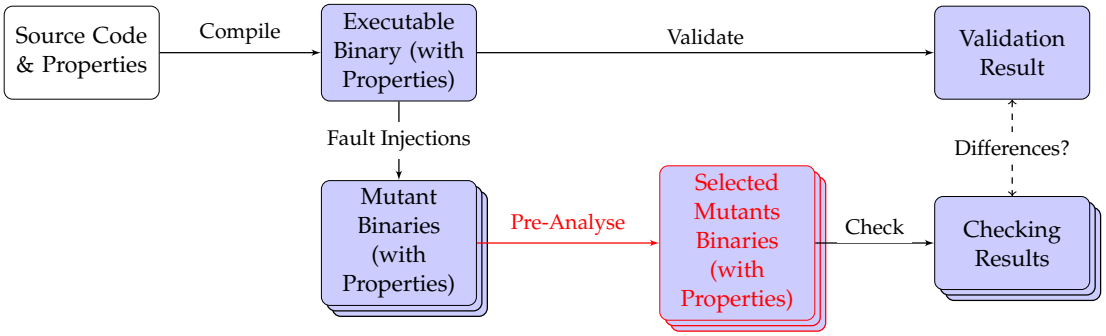

FIGURE 4.2 – Process Diagram with Pre-Analyse Step

Note

The choice to start the preparation with the source code and not the binary is made for illustrative clarity and ease of use for the software developer, since defining properties over binaries is more arduous. However, most aspects of the process do not rely upon this choice, and the improved version of the FIVD process starts directly from the binary (see in Section 9.1)

This thesis considers fault injection attacks upon the binary. It is thus necessary to compile the source code (and here properties) into an executable binary. This executable binary represents the software application that would be executed by the system in practice. Thus to simulate fault injection attacks on the actual system, the executable binary must be used in the simulation. For the process here the compilation must maintain the properties as annotations in the executable binary.

FIVD Process for Large Programs

This section discusses the extended version of the FIVD process used to detect vulnerabilities. In section 4.1, the FIVD process was presented in detail, here a pre-analysis step is added to improve the efficiency of the process, specially for larger program that will require a lot of time for model checking.

The process presented in this section differs with the process presented in section 4.1 only by the pre-analysed step. An overview of the (extended) process is as follows. Prior to starting the process, the source code, and the properties represented by annotations within the source code, must be defined. The preparation step for the process is then to compile the source code and properties to produce an executable binary in a manner that preserves the properties as annotations. The properties are validated to hold on the executable binary using model checking. The executable binary is then injected with simulated faults to produce mutant binaries. Pre-analysis is used to filter out mutants that fail to execute, or fail to yield an output. Mutants that execute and produce an output are then passed to model checking. A difference in the results of validation and checking the properties indicates a vulnerability to the simulated fault injection. An illustration of the process is given in Figure 9.1 with the new extensions here shown in red. The rest of this section focuses on discussing the point of difference between this process and the previous process shown is section 4.1.

```
bool grantAccess = false;                                    1
bool badValue = false;                                       2
int i = 0;                                                   3
while (i < PINSize) {                                        4
  if (PINCandidate[i] != PINTrue[i]) {                       5
    badValue = true;                                         6
  }                                                          7
  i++;                                                       8
}                                                            9
if (badValue == false) {                                     10
  grantAccess = true;                                        11
}                                                            12
```

FIGURE 4.3 – Motivating Example Code

Pre-analysis is performed on the mutant to quickly eliminate mutants that fail to execute or fail to produce an output. Pre-analysis includes detection of: crashes, infinite loops, and error codes. This saves model checking effort when there is already evidence that the properties will fail to hold.

The pre-analysed step is added just after the fault injection simulation step which can have the following effects on the executable binary:

— The simulation of fault injection upon the executable binary can produce mutant binaries which does not respect the binary structure, so the mutant binary is no more valid for execution and will crash.

— The simulation of fault injection upon the executable binary can change the loop condition in the produced mutant binary, and might create infinite loops, in this case the binary will infinitely loop through itself and never terminate.

— The simulation of fault injection upon the executable binary can also change an instruction in binary in a way that it will replace it with unknown instruction which might lead the program to crash.

In all the cases discussed bellow the produced mutant binary will not be valid to continue in the FIVD process. Since in the checking step, the process will be verifying invalid binary, which cause the model checking to crash. Model checking time is the important time compared to the other steps of the process. In the case of large programs like the cryptographic algorithm implementation, the objective will be to gain the maximum time we can in order to do the verification at the minimum of time.

Mutants that pass pre-analysis are then checked using model checking. Differences in the properties between the validation and the checking indicates that the fault that was injected yields a change in behaviour that violates the properties, and so could be exploited by an attacker.

4.2 FIVD Process Implementation

This section gives an overview of the FIVD process implementation. For more details about the FIVD process implementation please refer to the Appendix A where detail information is given about each step.

The implementation here exploits existing open source tools, despite some having limitations. This choice was made in order to focus upon a simple and feasible implementation of the process as a proof of concept. Discussion about the limitations of tools used in the implementation is in Section 7.1.

An overview of the implementation is as follows, and shown in Figure 4.4. The implementation begins with the source code written in the C language and the properties represented in the source code by assert statements. The source code and properties are compiled to an executable binary by the GNU C Compiler (GCC) [57]. The executable binary (including the properties contained within) is transformed into an *intermediate representation* in Low Level Virtual Machine Intermediate Language (LLVM-IR) by the Machine Code Semantics (MC-Sema) tool [136]. The properties are then checked on the intermediate representation using the Low Level Bounded Model Checker (LLBMC) [93, 129].

The fault injection is simulated on the executable binary using the SimFI tool in order to produce mutant binary files according to the chosen fault model. The steps to model check the properties on the executable binary are then repeated for the mutant binaries. Finally, the results of model checking the executable binary and the mutant binary are compared for differences.

More detailed

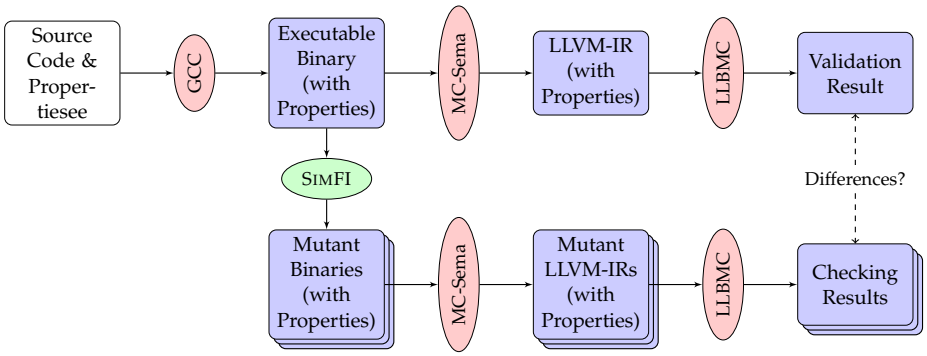

FIGURE 4.4 – Implementation Diagram

FIVD Process Implementation for Large Programs

This section presents the implementation of the extended version of the FIVD process. The implementation presented here differs from the one presented in the section above by the additional pre-analyse step. This section will focus on the pre-analysed step added to the FIVD process.

An overview of the implementation is as follows, and shown in Figure 4.5, the added extensions is shown in red. The implementation begins with the source code written in the C language and the properties represented in the source code by tool specific annotations. The source code and properties are compiled to an executable binary by the GNU C Compiler (GCC) [57]. The executable binary (including the properties contained within) is transformed into an *intermediate representation* in Low Level Virtual Machine Intermediate Language (LLVM-IR) by the Machine Code Semantics (MC-Sema) tool [136]. The properties are then checked on the intermediate representation using the Low Level Bounded Model Checker (LLBMC) [93, 129]. The SimFI tool is used to automatically generate the mutant binaries by injecting faults into the executable binary according to the chosen fault model. Pre-analysis is then

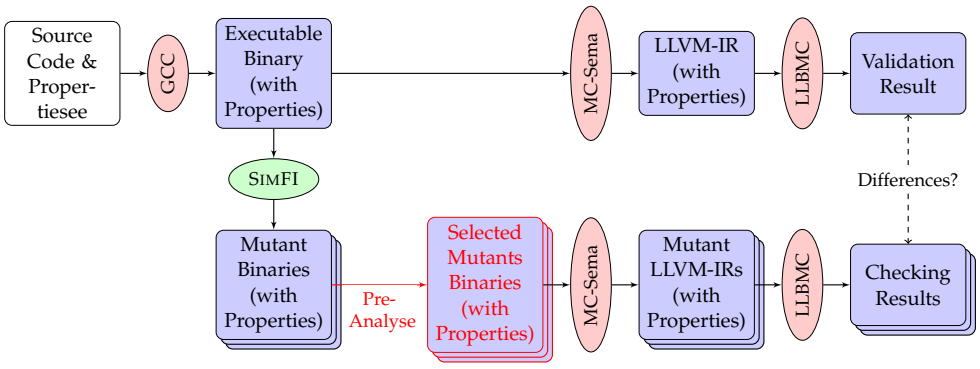

FIGURE 4.5 – Implementation Diagram with Pre-Analyse Step

performed that executes the mutant binaries and checks for various failures or non-output states. If pre-analysis is passed, then the mutant binary is transformed via MC-Sema and LLVM-IR to be checked by LLBMC as above. Finally, the results of model checking the executable binary and each mutant binary are compared for differences by matching the outputs of LLBMC.

The pre-analysis step is done using a script shell. The list of generated mutant binaries are given as an input. The script loops through the list of mutant binaries, executes each mutant binary, then based on the output result classify the mutant binaries. If the mutant binary is executed with no error, the mutant binary is saved as a valid mutant binary for the next step which is the checking step. If the mutant binary execution outputs an error it is classified as well in order to understand why the mutant binary is not executed. The mutant binary execution failure can be:

1. Related to an infinite loop which is created by the fault injection. The mutant binary is considered to have an infinite loop if the execution does not end after five seconds of execution.
2. Caused by illegal instruction set. The injection of fault model can modify the binary in way that the modified binary will correspond to an unknown instruction set.
3. Due to an aborted instruction or segmentation fault or other errors. Based on the information given above one can conclude that the simulation of fault injection can cause a variety of failure in the mutant binary.

4.3 FIVD Process Methodology

Experiment Design

Each experiment fixes one of the six fault models, and one of the three properties for the chosen encryption algorithm. Then the process is applied, testing all possible mutations that can be produced by the fault model, applied to the binary for the encryption algorithm, and testing for the property chosen. This yields thirty-six different experiments (although results are merged in following sections).

Each experiment was run on a twin Intel Xeon E5-2660 with 2×14 cores (maximum of 56 parallel threads) with 128GB of memory. The operating system is Ubuntu 16.04 (kernel version 4.4.0-21). The experiments are limited to a maximum of 17 parallel instances since LLBMC uses approximately 7GB of memory, and thus this prevents

any possibility of memory exhaustion. Multiple experiments were run in parallel on identical hardware instances.

Runtime Information

In the experiments, model checking was the part that had major impact on the execution time of our process. Note that we managed to parallelise the experiments. This makes it hard to compute the exacte execution time [1], but speed up the process.

However, the experiments for the PRESENT algorithm for Property 1, using the FLIP fault model, required the following time resources. The total number of mutants that was generated after the fault injection situation step is 9192 mutants, and the number of selected mutant to model check is 4797. To model check the selected mutant `3days 18hours 22minutes` were required. Which gives an approximation of `1min 13s` for a mutant to be model checked by LLBMC.

Fault Models

This section details the fault models that are used in the experiments later on.

The FLIP *fault model* simulates the flipping of a single bit, either from 0 to 1 or from 1 to 0. This fault model is highly representative of many kinds of faults that can be induced, ranging from those due to atmospheric radiation, to software effects such as the rowhammer attack [76].

The Z1B *fault model* simulates setting a single byte to zero (regardless of prior value). This fault model reflects a more malicious attack in general, and corresponds to a commonly achievable attack in practice [123, 137].

The Z1W *fault model* represents setting an entire word to zero (again regardless of prior value). This is similar in concept to the Z1B fault model and attack, but captures behaviour more related to the hardware model, since it reflects faulting some piece of the hardware that operates on words rather than bits or bytes (such as the bus).

The NOP *fault model* sets the targeted operation to a non-operation instruction for the chosen architecture (in this case 0x90 for X86 architecture). The concept behind this model is that it simulates skipping an instruction, a common effect of many runtime faults [97]. However, due to the inconsistent alignment of instructions in X86 this fault model may also change parts of other instructions or values when the alignment does not match.

Both the JMP *fault model* and the JBE *fault model* simulate faulting the target address of a jump instruction. In practice there are multiple jump instructions, either unconditional jumps JMP or conditional jumps JBE. In both cases, the fault model simulates any possible change to the jump target address. This fault model corresponds to targeted attacks that attempt to bypass code, or significantly disrupt the control flow [46, 110]. Note that this is a superset of other faults that may target the same bits of the target address.

1. The fact that the experiments were conducted in parallel in different instances, and because the experiments needed to be restarted due to problems on the configuration of the server. It was hard to provide consistent runtime information for all the experiments.

Chapter 5

Experimental Results

This chapter investigates the efficiency and the scalability of the FIVD process by applying it to larger and real implementations. This chapter first demonstrates how vulnerabilities are detected on the motivating example using the FIVD process. Then, the FIVD process is applied to the implementations of two cryptographic algorithms, PRESENT and SPECK, to show the efficacy and scalability of the process.

Sommaire

5.1 Motivating example

This section presents experimental results obtained by applying the FIVD process on the motivating example. For clarity, all the attacks and properties presented here use the motivating example.

The experimental results show that by using the FIVD process a variety of fault injection vulnerabilities can be detected. The rest of this section presents illustrative 1-bit fault injection attack examples that illustrate how changing a single bit in a binary file can lead to a vulnerability in the program behaviour. A general solution that can detect many kinds of fault injection attacks will be presented, showing how easy is it to automate the FIVD process. The section will conclude with a variety of different kinds of fault injection attacks to illustrate the generality of the process and implementation.

Attack 1

Considering the motivating example in Figure 4.3, the first attack to consider is the attack where changing a single bit changes the `PINSize` variable from `4` to `0`.

Notice that in the motivating example source code, the loop that checks the digits of the PINs iterates from `i = 0` to `i < PINSize`, that the loop will be skipped (since `0 < 0` does not hold). If an attacker succeed in flipping a single bit that will change the value of `PINSize` variable from `4` to `0`, a vulnerability will be created since the PINs are never checked against each other.

Therefore, any candidate PIN will lead to access being granted, and so the attacker can use this fault injection attack to gain access (even when the candidate PIN they supply does not match the true PIN).

Property A

A simple property to detect such an attack would be to ensure that `i` reaches `4`. This can be achieved by taking the following property:

```
__llbmc_assert(i == 4);
```

inserted between lines 9 and 10. This property checks if after the loop the value of the variable `i` is `4`

With this property added to the source code the process was repeated with attack 1. The results of model checking the mutant binary reveal that the assertion was violated and thus the model checking result differs form the validation result. Thus, vulnerability to the first fault injection attack is detected.

Attack 2

An alternative 1-bit fault injection attack that has the same effect as Attack 1 is to initialise the value of the variable `i` to `4`. This will again grant access even when the two PINs differ since the loop will be bypassed as before (this time since `4 < 4` does not hold). Observe that since `i` is initialised to `4` this fault injection attack should not violate the assert statement of Property A. This can be exploited by the attacker in the same manner as Attack 1 to gain access with a candidate PIN that does not match the true PIN.

The process was repeated with Attack 2 and Property A. As expected, the fault injection vulnerability was not detected. Property A was not able to detect this fault injection attack (simulation).

Property B

The above result illustrates that the choice of properties needs to consider the behaviour of the program rather than focus on particular variables that are incidental to the program's execution. Thus, a property that captures the idea that unequal PINs should never lead to access being granted could be defined as follows.

```
__llbmc_assert(    !(PINCandidate != PINTrue)
                || grantAccess == false);
```

where this property is inserted into the motivating example code after line 12. This property expresses that if the two PINs are different then the access is not granted.

The process was then repeated for both Attack 1 & 2, and with Property B. As expected Property B was able to detect both fault injection vulnerabilities represented by Attacks 1 & 2. This shows that **considering the behaviour is more important than considering the variables used to achieve the behaviour**. That is, properties should consider `PINCandidate`, `PINTrue`, and `grantAcces` rather than `i` or `badValue`.

Attack 3

Observe that Property B detects attacks that allow access when the PINs are not equal, but does not consider when the PINs are equal. An alternative attack could be to deny access even to a user who knows the correct PIN. Consider the 1-bit fault

injection attack that changes the value of `true` (represented in binary by `0...01`) to `false` (represented in binary by `0...00`).

The objective of this attack is blocking authorised access rather then allowing unauthorised access as in Attack 2.

Now consider this attack upon the motivating example line 11, changing `grantAcces = true` to be

```
grantAccess = false;
```

and thus preventing any access even when the PINs are equal.

The process was then repeated using this new Attack 3 with Properties A & B. As expected neither property was able to detected this attack. Property A failed since the attack does not effect the iterator `i`. Property B failed since when the PINs are equal no further behaviour is considered.

The pecification of a property that will capture the general wanted behaviour from the program is needed.

Property C

To also account for Attack 3, the original Property B needs to be extended to also consider the behaviour when the PINs are equal. Indeed, the ideal behaviour of the code can be represented by the following property.

```
__llbmc_assert(
     (   !(PINCandidate != PINTrue)
      || grantAccess == false)
  && (   !(PINCandidate == PINTrue)
      || grantAccess == true));
```

This ensures that when the PINs are unequal access is not granted, and when the PINs are equal then access is granted. Property C is added to the motivating example in the same place as Property B would be; after line 12.

The process was then repeated with all three Attacks (1, 2 & 3) using Property C. As expected Property C was able to detect all three fault injection attacks.

Property C succeed in capturing all the three attack because it was specified to capture the behaviour of the program. Contrary to the previous properties (A and B), property C does not consider only single variable value, or a single behaviour of the program, but expresses the general behaviour of the program.

Other Attacks

This section considers several more fault injection attacks in less detail than those above. These include several more 1-bit fault injection attacks, and then other kinds of attacks, culminating in an attack that can only be effected in the binary and not in the source or by "compiling" directly to an intermediate representation.

There are several other 1-bit fault injection attacks that can be performed against the motivating example. Such attacks include changing the initialisation value of variables such as `grantAccess` and `badValue`, e.g. at line 6 changing the initialisation

`badValue = true` to instead be `badValue = false`. These are all detected by at least Property C, if not also Property B.

A different kind of attack that targets the control flow of the program is to change the target of a jump instruction. For example, the jump from the conditional on line 5 of the motivating example could change from skipping the following instruction on line 6, to always executing line 6. Thus `badValue = true` (line 6) would always be executed, regardless of the outcome of the conditional. This can be done by modifying three bits (or one byte) of the target (relative) address of the jump, from `0000 0111` to `0000 0000`. This attack was successfully detected by Property C (but not Properties A or B).

A more significant attack on the behaviour of an instruction is to simply change the instruction to a NOP (non-operation). Consider in the motivating example when a fault injection changes the instruction that represents line 6 (Figure 4.3) `badValue = true;` to a NOP. This requires modifying 4 bytes (on the X86 32-bit architecture used here). The change would allow access for any candidate PIN (by never recording differences to `badValue`). This attack was successfully detected by Properties B and C (but not Property A).

Instead of changing a subtle behaviour like a jump, or simply wiping an instruction to a NOP, another fault injection attack is to change the instruction type, e.g. changing a `CMP` instruction to a `MOV` instruction. This can be done on the `CMP` instruction that compares the values `PINCandidate[i] != PINTrue[i]` on line 5 of the motivating example. This requires modifying 3-bits of the executable binary, from `0011 1011` to `1000 1011`. The result of this change is that the following line that sets `badValue` to `true` will always be executed. This change prevents access even when the correct candidate PIN is provided, similar to Attack 3. As expected, this attack is successfully detected by Property C (but not Properties A or B).

One attack that is of particular interest here, is an attack that can be represented in the executable binary but not in source code or from "compiling" the source code to an intermediate language, such as C and LLVM-IR, respectively. An example of this kind of attack is the modification of the return value stored in the return register (`eax` on X86 architectures). This kind of attack can be simulated and detected using the process and implementation here.

To properly handle this attack requires a function call, and so the motivating example is modified by placing the code in Figure 4.3 inside a function that returns `grantAccess`. The attack works by altering the returned value from this new function. The return value is stored in the `eax` register, for the motivating example this is handled by the binary operation corresponding to the assembly instruction below.

```
mov    eax, DWORD PTR [ebp-0x8]
```

The attack is then to change the value loaded into `eax` so that the function behaviour is changed. For example, by changing the value loaded into `eax` from `0000 1000` to `0000 1100` the returned value of `grantAccess` can change from `False` to `True`, so the access will be granted even if the two PINs are not equal. (Note that if the returned value is already `0000 1100` then this can be ignored, or the value changed to `0000 1000` inverting the function behaviour). This attack is detected by Property C, although Property C needs to be located outside the function call so as to check the value of `grantAccess` after the function return (or more precisely the returned value that corresponds to `grantAccess`).

These attacks illustrate that the process and implementation here are not limited to a single fault model or kind of fault injection attack. Thus, the same process and implementation can be used for a variety of fault models and fault injection attacks, as long as some consideration is taken to choose the right properties. Further, the process and implementation here can detect fault injection attacks that cannot be found by checking the source code or intermediate representation alone; the executable binary must be part of the process and implementation.

Computational Data

This section discusses the experimental results of automating the process and implementation of this part. This automation is straightforward once all the tools are available.

To support the automation experiments, a SimFI fault injection tool was created. This tool takes an executable binary, and produces mutant binary based on the chosen fault model. For illustrative reason, we fixed the fault model for this experiment to be zeroing one byte, this fault model consists at replacing one byte with 0. This fault injection model is naive, but simple to implement and conduct experiments to test the automation of the process and implementation.

To estimate the feasibility of finding arbitrary fault injection vulnerabilities in the executable, the following experiment was conducted. A script was written that takes a source code and properties, and compiles these to an executable binary with GCC. This executable binary is then validated to ensure that the properties hold. The SimFI fault injection tool was used to generate mutant binaries for each possible byte in the executable binary being set to 0. The script then enters into a loop over the mutant binaries that generates the LLVM-IR for the mutant binary and model checks the properties on this LLVM-IR. The result of this model checking is then used to determine if the injected fault creates a vulnerability. The runtime and number of fault injection vulnerabilities was counted and reported at the end of the experiment.

This script was run over a version of the motivating example (Figure 4.3). The fault injection tool was limited to injecting faults into the `.text` area of the executable binary that corresponds to the compiled source code (to reduce time wasted model checking fault injection vulnerabilities in the header or other unrelated parts of the executable binary).

The executable binary produced by GCC in this case was 3024 bytes. The `.text` area was 159 bytes and thus 159 1-byte fault injection attacks were simulated, yielding 159 mutant binaries. The following experiments were conducted on a virtual machine configured with one CPU, and 7662 MB of RAM running Linux Ubuntu 14.04 LTS. The virtual machine was hosted on a Macbook Pro with 3,1 GHz Intel Core i7 processor, 16 GB of RAM, running OS X EL Capitan 10.11.

Three different experiments were conducted, testing the three different properties presented earlier. An overview of the result is given in Table 5.1. The first experiment with Property A detected 36 fault injection vulnerabilities, and had a runtime of 7 minutes. The second experiment with Property B detected 37 fault injection vulnerabilities, and had runtime of approximately 1 hour. The third experiment with Property C detected 37 fault injection vulnerabilities, and had a runtime of approximately 2 hours. The main cost in time was the model checking by LLBMC, as is clearly shown by the significant difference made by the choice of property. The results showed that the specified property had a direct impact on the required time.

	PRESENT						SPECK					
	FLIP	Z1B	Z1W	NOP	JMP	JBE	FLIP	Z1B	Z1W	NOP	JMP	JBE
Model Check	4797	411	3	400	782	613	4199	492	38	316	230	83
Infinite Loop	464	44	0	63	545	69	80	11	1	7	72	0
Illegal Instruction	473	5	0	55	26	63	315	2	1	38	16	8
Aborted	355	23	2	49	187	141	110	6	4	8	14	8
Segmentation Fault	3099	666	1144	589	1263	1639	3525	519	986	659	429	155
Other Errors	4	0	0	0	2552	2830	11	0	0	2	259	766

TABLE 5.1 – Overview of Fault Injection Pre-Analyse Results

Observe that the number of fault injection attacks to test in this manner is linear in the size of the binary executable. Thus, automatically testing all such fault injection attacks is feasible, particularly since the implementation can be easily run in parallel.

Property	Vulnerabilities Detected	Runtime
Property A	36	7 minutes
Property B	37	1 hour
Property C	37	2 hours

FIGURE 5.1 – Experimental Results for the Motivating Example

5.2 Cryptographic Algorithm

This section presents the results of experiments on PRESENT and SPECK algorithms. This includes the results of both the pre-analysis and the model checking. The experiment design is presented first, then overviews of pre-analysis and model checking, and finally vulnerabilities are discussed in PRESENT and SPECK.

For the conducted experiments three properties were considered that can be applied to the encryption algorithms.

Property 1 is to check whether the encryption was conducted successfully, i.e. the ciphertext at the end of the encryption function corresponded was correct. This allows the detection of any kind of damage to the encryption algorithm by fault injection. However, further properties are considered to examine more specific vulnerabilities.

Property 2 checks whether the ciphertext is equal to the plaintext, i.e. the encryption can be bypassed by fault injection. This is an extreme vulnerability where a fault is able to entirely skip the encryption and render the binary useless (albeit still appearing to execute without error state and produce a "ciphertext").

Property 3 is specific to each encryption algorithm, and results in a key recovery attack when combined with a cryptanalytical attack (CA). The CAs are: for PRESENT skipping the last (31st) round [52, 144], and for SPECK to output the result of implementing only 9 or 10 rounds [42].

Pre-Analysis Results

The result of the pre-analysis can be seen in Table 5.1. The effect of the different fault models is fairly consistent on both algorithms.

The FLIP fault model yields many different behaviours, but the vast majority still produce output that needs to be model checked to determine the effect. Infinite loops, illegal instructions, and aborted results from execution are common, but all

		PRESENT						SPECK					
		FLIP	Z1B	Z1W	NOP	JMP	JBE	FLIP	Z1B	Z1W	NOP	JMP	JBE
Prop. 1	Good Ciphertext	1080	52	0	40	96	82	1536	205	1	86	10	0
	Bad Ciphertext	2249	116	1	173	532	362	2514	264	31	188	216	83
	Crashed	1471	239	2	186	154	169	149	24	6	42	4	0
Prop. 2	Leak Plaintext	0	0	0	0	1	8	3	0	0	2	2	2
	Not Leak Plaintext	3285	162	1	201	439	445	671	294	17	224	88	2
	Crashed	1472	246	2	190	153	91	3433	175	8	64	4	0
Prop. 3	CA Vulnerable	48	0	0	5	10	1	0	0	0	0	0	0
	Not CA Vulnerable	3247	163	1	199	430	447	2244	401	17	255	90	4
	Crashed	1466	245	2	186	152	90	1860	49	8	65	4	0

TABLE 5.2 – Overview of Fault Injection Model Checking Results

minority results. Segmentation faults are highly likely, but not as likely as an output. Other errors are very rare.

Both the Z1B and NOP fault models were highly likely to cause a segmentation fault. The next most likely outcome was an output that required model checking. Infinite loop, illegal instruction, and aborted were very rare, and other errors extremely rare (only evident in SPECK). The similarity of results for these fault models is not surprising, since they both change the value of a byte in the same manner.

The Z1W fault model almost always caused a segmentation fault. All other outcomes were extremely rare or non-evident. In particular, this meant that very few required model checking and so the pre-analysis was highly effective at reducing the cost here.

The JMP and JBE fault models predominantly produced other errors or segmentation faults, although a large number of mutants still required model checking. Many segmentation faults are expected since X86 has variable size instructions and often jumps target the middle of instructions, yielding invalid instructions or arguments.

Observe that the pre-analysis reduced the total number of mutants requiring model checking from 110157 to 37089, thus reducing the model checking effort by 66.33%.

Model Checking Overview

An overview of the results of model checking can be seen in Table 5.2. Observe that the most interesting result of each property has been highlighted in the table. The rest of this section overviews these results in a broad sense, while each encryption algorithm is discussed in detail in the following sections.

The validity of the model checking results was manually verified by randomly selected samples from the outcomes in Table 5.2. Several [1] mutants were checked from each combination of: encryption algorithm, property, and fault model. This resulted in manual verification of the results for 265 mutants to ensure the results were correct.

The evidence from Property 1 suggests that the investigated implementations of both PRESENT and SPECK are somewhat resistant to fault injection attacks, with approximately 25.78% of mutants still yielding correct output. Conversely 74.22%

1. Between 10 and 15 mutants, or all mutants if the total was less than 10.

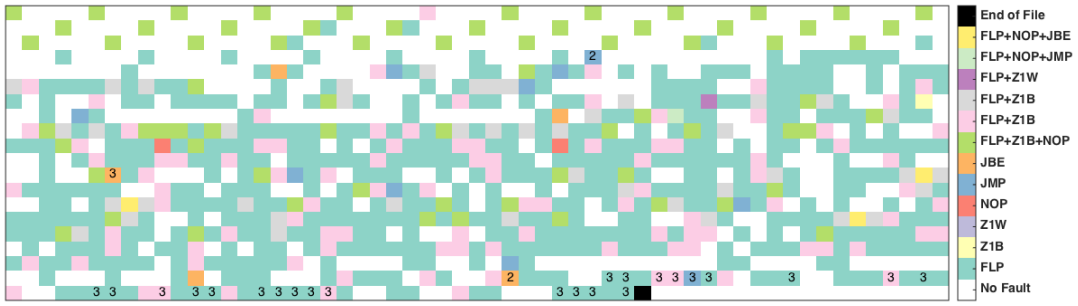

FIGURE 5.2 – Model Checking Results for PRESENT

produced some kind of output, usually a faulty output that does not match the vulnerabilities considered here [2]. Although Property 1 is specified to identify any incorrect output, the correct outputs are those of most significance here due to overlap between "Bad Ciphertext" and the other two properties.

Property 2 identifies the most severe outcome considered here; when the encryption algorithm appears to operate correctly but outputs the plaintext. Observe that there are very few fault injection attacks that can completely bypass the encryption algorithm, only 9 for PRESENT and 9 for SPECK. Detailed discussion of these follows in later sections.

Property 3 identifies specific Cryptanalytic Attacks (CA) vulnerabilities that an attacker can use to learn information about the key [24, 144]. These tend to be specific to the behaviour of the algorithm considered, e.g. skipping a specific iteration of a loop, yet still significant to security. These turned out to be achievable frequently (64 occurrences) and with most fault models upon PRESENT, but not achievable with any of the fault models considered here on SPECK.

Observe that several crash results occurred during model checking. These are due to various different outcomes, all listed as "Crashed" in Table 5.2. In 8.08% of cases these crashes were due to MC-Sema failing to parse the mutant into LLVM-IR. The vast majority, 91.9%, were due to failures of LLBMC, in particular segmentation faults in the SMT solver (STP with MiniSAT). Lastly, in the remaining 0.02% of cases LLBMC was unable to produce reliable output, stating that multiple mutually exclusive properties held [3].

PRESENT Vulnerabilities

This section explores the 73 vulnerabilities found in PRESENT. Only a small number of these (9 occurrences) were found to violate Property 2 and so yield the plaintext. The majority (64 occurrences) were CA vulnerabilities that allow the key to be calculated from a number of ciphertexts.

All Property 2 vulnerabilities occurred from modifications to jump instructions in the code, 1 for JMP and 8 for JBE. In practice these all succeeded by changing the

2. This does not preclude these outputs corresponding to a different vulnerabilities such as [42, 105].

3. This is clearly a limitation of LLBMC and was not attempted to be rectified here. Despite LLBMC being bounded, these results did *not* indicate the bound was reached. Many of these results were manually verified and found to only hold true for Property 1, i.e. produce a bad ciphertex.

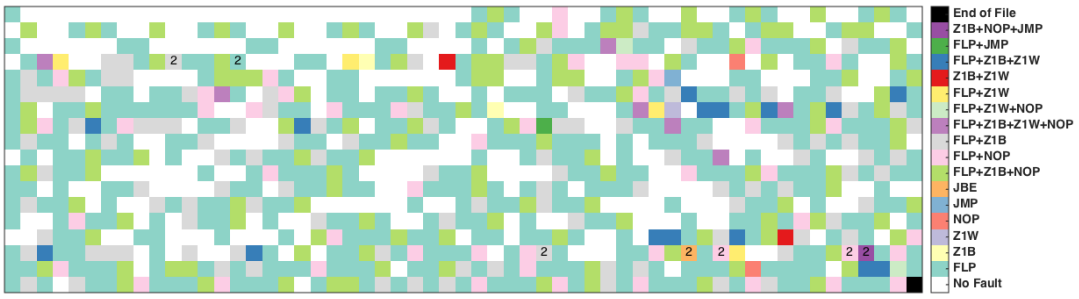

FIGURE 5.3 – Model Checking Results for SPECK

control flow to skip the entire algorithm and appear in only 2 places in the binary labeled with "2" in Figure 5.2.

The JMP occurs early in the code as a wrapper around the main encryption algorithm, and can be exploited to jump past the encryption. There is only 1 vulnerability here because any earlier target would still execute some part of the encryption (or crash), and any later target would jump beyond the limits of the code.

The JBE vulnerabilities are all from the same address, albeit there are multiple targets that the jump can be modified to and still yield successful execution. This jump is from the comparison for the first loop, and can instead be modified to jump past the first and second encryption loops (almost to the end of the execution). There are multiple targets here since there are several places in the last loop of the algorithm that do not change the output (and the conditional for exiting the loop holds due to incorrect stack pointer offsets). There are also targets beyond the end of the loop, although these can lead to program crashes depending on the target address.

Property 3 vulnerabilities, labeled "3" in Figure 5.2, are more common and can be achieved with a larger variety of fault models. Like the JMP of Property 2, the JBE here has a single address that can be jumped to yield a CA vulnerability. There are however multiple JMP vulnerabilities, all derived from changes to the exit jump of the main encryption loop being re-targeted to skip over the last round of encryption.

The FLIP and NOP vulnerabilities to Property 3 are all located towards the end of the binary, since this is where the last round of encryption is executed. The majority of these are minor changes to various instructions that prevent the last round being properly executed. These include: modifying the loop index in the last round (e.g. from 0 to 32), changing registers (e.g. loading to/from `%eax` instead of `%ecx`), changing values (e.g. changing while loop bound from > 7 to > -7), changing the comparison instruction to something else (e.g. `cmp` to `sub`), etc.

SPECK Vulnerabilities

This section explores the 9 vulnerabilities found in SPECK. All of these were found to violate Property 2 and so yield the plaintext. The lack of Property 3 vulnerabilities is discussed below.

The Property 2 violations were obtained with 4 fault models, FLIP, NOP, JBE, and JMP. The FLIP vulnerabilities occurred due to: 1 damaging the stack pointer and so

avoiding loops and skipping the whole algorithm, the other 2 changed the round key to prevent the encryption acting effectively.

The NOP vulnerabilities occurred indirectly by inserting the value for a NOP instruction (0x90) into another instruction. The first, in the jump instruction for the key generation loop and so skipping past the key generation. The second, in the encryption loop and so initialised the loop iterator to 0x90, immediately skipping the encryption.

The JBE and JMP both skip the encryption loop by jumping forward to the same 2 locations after the encryption has completed. The JBE is at the end of key generation, and the JMP at the beginning of the encryption.

Property 3 vulnerabilities were not found with any of the fault models tested here. This is unfortunate but not surprising since the implementation does not have simple code path that can be exploited to yield this particular vulnerability. (Unlike PRESENT where the last round is executed in a separate loop.)

The most likely candidate fault model would have been FLIP where some simple flip of a variable or value could have altered the number of encryption rounds executed. Unfortunately, since 21 rounds are executed and CA vulnerabilities have been published for 9 or 10 rounds, the numbers 21 and 9 or 10 are not a single bit flip apart. However, to validate the experiments, a manual fault injection of setting the loop bound value to 10 was tested and found to be CA vulnerable with the process.

Chapter 6

Details on the Experimental Results and Implementation for the PRESENT Algorithm

This chapter demonstrate in details how the results presented in Chapter 4 were obtained. By following the steps of the extended version of the FIVD process presented in Chapter 5. This chapter takes the PRESENT algorithm as an example, shows how the properties are first specified on the C source code implementation. How the results were returned. Then, how the obtained results were verified, to check if the obtained results were correct, and can be explained.

Property Specification

Listing 6.1 presents the PRESENT algorithm C implementation used in Chapter 4. Observe that the properties were asserted from line 150 to 160 in the Listing 6.1. The properties correspond to the three verified ones in Chapter 4.

— *Property 1* checks if the encryption was conducted successfully, i.e. the ciphertext at the end of the encryption function corresponds to the correct one.

```
__llbmc_assert( (state[0]==0xcc) && (state[1]==0x71) && (state
    [2]==0x59) && (state[3]==0x6c) && (state[4]==0x15) && (
    state[5]==0x58) && (state[6]==0xcd) && (state[7]==0x47));
```

— *Property 2* checks if at the end of the encryption function the ciphertext is equal to the plaintext, i.e. the encryption can be bypassed by fault injection.

```
__llbmc_assert( (state[0]!=plaintext[0]) || (state[1]!=
    plaintext[1]) || (state[2]!=plaintext[2]) || (state[3]!=
    plaintext[3]) || (state[4]!=plaintext[4]) || (state[5]!=
    plaintext[5]) || (state[6]!=plaintext[6]) || (state[7]!=
    plaintext[7]) );
```

— *Property 3* correspond to a cryptanalytical attack (CA) [52, 144] that can recover the key, i.e. It is possible to skip the last (31st) round by fault injection.

```
__llbmc_assert((state[0]!=0x17) || (state[1]!=0x89) || (state
    [2]!=0xe6) || (state[3]!=0xf4) || (state[4]!=0xb9) || (
    state[5]!=0xc8) || (state[6]!=0x95) || (state[7]!=0x43));
```

The variable `state[]` in this implementation correspond to the ciphertext, this variable contains the value of the ciphertext through the encryption process. The variable `plaintext[]` contains the plaintext value, specified in the start of the implementation line 22 in Listing 6.1.

For the rest of this chapter, the focus will be on *Property 2*. It is an interesting vulnerability, that will allow the attacker to get the plaintext, without needing the key. The following sections will presents how the results were obtained, and then how it is possible to understand the obtained results.

```
1  /*****************************************************************************
2  Written and Copyright (C) by Dirk Klose
3  and the EmSec Embedded Security group of Ruhr-Universitaet Bochum.
4  All rights reserved.
5
6  Contact lightweight@crypto.rub.de for comments & questions.
7  This program is free software; You may use it or parts of it or
8  modifiy it under the following terms:
9
10 If you are interested in a commercial use
11 please contact '''lightweigth@crypto.rub.de'''
12 *****************************************************************************/
13
14 // Include-file
15 #include<stdint.h>
16
17 int main(void)
18 {
19   const uint8_t sBox4[] = {0xc,0x5,0x6,0xb,0x9,0x0,0xa,0xd,0x3,0xe,0xf,0x8,0x4,0x7,0x1,0x2};
20 //  Input values
21   uint8_t key[] = {0x23,0x40,0xa2,0x63,0xb4,0x56,0x89,0x6b,0x31,0x6c};
22     uint8_t plaintext[] = {0x69,0x6b,0x86,0xa4,0x85,0x22,0x1b,0xc8};
23     volatile uint8_t state[8];
24 //  Counter
25  uint8_t outAssert[8];
26   uint8_t i = 0;
27 //  pLayer variables
28   uint8_t position = 0;
29   uint8_t element_source = 0;
30   uint8_t bit_source = 0;
31   uint8_t element_destination = 0;
32   uint8_t bit_destination = 0;
33   uint8_t temp_pLayer[8];
34 //  Key scheduling variables
35   uint8_t round;
36   uint8_t save1;
37   uint8_t save2;
38     uint8_t j=0;
39 //  ***************** Set up state *************************
40     for(j=0;j<8;j++)
41     {
42         state[j] = plaintext[j];
43     }
44
45 //  ***************** Encryption *************************
46   round=0; // change the value of the variable 'round' from O to 64 to skip the while loop
47   while (round<31)
48   //do
49   {
50 //  ***************** addRoundkey ************************
51     i=0;
52     while (i<=7)
53     //do
54     {
55       state[i] = state[i] ^ key[i+2];
56       i++;
57     }
58     //while(i<=7);
59 //  ***************** sBox *******************************
60     while(i>0)
61     //do
62     {
63       i--;
64       state[i] = sBox4[state[i]>>4]<<4 | sBox4[state[i] & 0xF];
65     }
66     //while(i>0);
67 //  ***************** pLayer *****************************
68     for(i=0;i<8;i++)
69     {
70       temp_pLayer[i] = 0;
71     }
72     for(i=0;i<64;i++)
73     {
74       position = (16*i) % 63;           //Artithmetic calculation of the pLayer
75       if(i == 63)                 //exception for bit 63
76         position = 63;
77       element_source    = i / 8;
78       bit_source      = i % 8;
79       element_destination = position / 8;
80       bit_destination   = position % 8;
81       temp_pLayer[element_destination] |= ((state[element_source]>>bit_source) & 0x1) << bit_destination;
82     }
83     for(i=0;i<=7;i++)
84     {
```

```
85        state[i] = temp_pLayer[i];
86      }
87  //  ****************** End pLayer ************************
88  //  ****************** Key Scheduling *********************
89      save1  = key[0];
90      save2  = key[1];
91      i = 0;
92      while(i<8)
93      //do
94      {
95        key[i] = key[i+2];
96        i++;
97      }
98      //while(i<8);
99      key[8] = save1;
100     key[9] = save2;
101     i = 0;
102     save1 = key[0] & 7;                 //61-bit left shift
103     while (i<9)
104     //do
105     {
106       key[i] = key[i] >> 3 | key[i+1] << 5;
107       i++;
108     }
109     //while(i<9);
110     key[9] = key[9] >> 3 | save1 << 5;
111
112     key[9] = sBox4[key[9]>>4]<<4 | (key[9] & 0xF);  //S-Box application
113
114     if((round+1) % 2 == 1)               //round counter addition
115       key[1] ^= 128;
116     key[2] = ((((round+1)>>1) ^ (key[2] & 15)) | (key[2] & 240));
117
118     if(round == 17){
119       for (i=0;i<8;i++){
120                       outAssert[i]=state[i];
121       }
122     }
123 //  ****************** End Key Scheduling *****************
124     round++;
125   }
126   //while(round<31);
127 //  ****************** addRoundkey ************************
128   i = 0; // change the value of the variable 'i' from 0 to 64 to skip the while loop
129   while(i<=7)
130   //do                      //final key XOR
131   {
132     state[i] = state[i] ^ key[i+2];
133     i++;
134   }
135   //while(i<=7);
136
137 //  ****************** End addRoundkey ********************
138
139 //  int x=0;
140 //  while (x < 8)
141 //  {
142 //    printf("0x%02x ",state[x]);
143 //    x++;
144 //  }
145
146 //  ****************** End Encryption  ********************
147
148 //      ****************** START Properties *********************
149
150 //Property 1: Equal correct ciphertext
151 //  __llbmc_assert( (state[0]==0xcc) && (state[1]==0x71) && (state[2]==0x59) && (state[3]==0x6c) && (state[4]==0
        x15) && (state[5]==0x58) && (state[6]==0xcd) && (state[7]==0x47));
152
153 //Property 2 : Plaintext equal ciphertext
154   __llbmc_assert( (state[0]!=plaintext[0]) || (state[1]!=plaintext[1]) || (state[2]!=plaintext[2]) || (state
        [3]!=plaintext[3]) || (state[4]!=plaintext[4]) || (state[5]!=plaintext[5]) || (state[6]!=plaintext[6]) ||
        (state[7]!=plaintext[7]) );
155
156
157 //Property 3 : Equal to the Last round
158 //0x17 0x89 0xe6 0xf4 0xb9 0xc8 0x95 0x43
159 //  __llbmc_assert((state[0]!=0x17) || (state[1]!=0x89) || (state[2]!=0xe6) || (state[3]!=0xf4) || (state[4]!=0
        xb9) || (state[5]!=0xc8) || (state[6]!=0x95) || (state[7]!=0x43));
160
161 //      ****************** END Properties *********************
162
163   return 0;
164
165 }
```

LISTING 6.1 – PRESENT C Implementation with properties

Obtaining the Results

As it was explained in Chapter 4, in order to obtain the results the FIVD process implementation was adopted (see Figure 4.5, page 46).

The bash script in Listing 6.2 shows in a summarised way the FIVD process implementation. The script takes the C code within the properties as input, then it generates the executable binary within the properties. The first step consists of validating the property upon the executable binary. If the property is validated, then the second step consists in generating the mutant binaries, by simulating the chosen fault model on the executable binary. The third step is the pre-analyse step, this step eliminates the mutant binaries that fail to execute, or fail to yield an output. The fourth step consists in model checking the mutant binaries that execute and produce an output which were passed from the third step.

The pre-analysed step here, executes the mutant binaries in order to classify them. The mutant binaries are run on the hardware. Note that, for this case study (of cryptographic block ciphers) only a single trace (single input) of the mutant binary is necessary. However, for other case studies with different information flows multiple traces (inputs) may be required in order to capture all possible branches.

```
#!/bin/bash

filenameC=$1  # The source code file of program to verify
size=$2       # The size of the program

#Step 1: Validation of the property + Step 2 : Generate Mutant if
    validate
./Validation-GenMutant.sh $filenameC $size

#Step 3: Preparation before the verification
./PrepareMutant.sh $filenameC $size

#Step 4: Verification of the selected mutants using model checking
./VerifOnlyNoCrashMutant.sh
```

LISTING 6.2 – Bash Script

Results

Listing 6.3 shows a possible output when running the script in Listing 6.2. The given information will allows tracking the state of the experiments.

```
The verification may take few minutes
The property holds on the binary the generation and verification of
    mutants will start
The mutants are generated
Pre-Analyse Step is done
Start the verification
The verification may take few minutes
The verification may take few minutes
test mutant number 4
The verification may take few minutes
test mutant number 2
test mutant number 1
The verification may take few minutes
test mutant number 3
...
```

LISTING 6.3 – Terminal Output

The results of the experiments are stored in files, which are labeled based on the verified property and the chosen fault model. Listing 6.4 presents a part of a file where, the results of the pre-analyse step for the PRESENT algorithm using the FLIP fault model, are stored. Each line in the Listing 6.4 gives two information:

— the mutant binary number, which is used to indicates the address of the byte that was modified, and

— the result of the pre-analyse step.

If we take as an example the first line in Listing 6.4: `52-0,segFault`. `52-0` corresponds to the number of the mutant, but it also indicates the the address of the bit that was modified by the fault injection simulation, so here the fault injection simulation modified the bit 0 in the byte 52. `segFault` correspond the pre-analysed step result for the mutant `52-0`, here it indicates that the mutant fails to execute due to a segmentation fault.

```
52-0,segFault
52-1,segFault
52-2,segFault
52-3,segFault
52-4,segFault
52-5,succeed - Good cipher text
52-6,succeed - Good cipher text
52-7,succeed - Good cipher text
53-0,succeed - Good cipher text
53-1,segFault
53-2,segFault
53-3,segFault
```

LISTING 6.4 – The Pre-Analyse Step Results For Property 2 Using FLIP Fault Model

The selected mutants after the pre-analyse step are then verified. Listing 6.5 presents a part of a file in which the results of the verification of *Property 2* on the PRESENT algorithm using the FLIP fault model are stored. Similarly to Listing 6.4, each line in Listing 6.5 gives the mutant number and the result of its verification. Here `Verified` indicates that the property holds, which means that the ciphertext was not equal to the plaintext, so in this case the fault injection did not create a vulnerability. `Violated` indicates that the property was violated, this means that the fault injection created a vulnerability. In the other cases with output `Crashed` or `CFG not generated` the verification was not performed due to problems in the model checker.

```
52-5,Verified
52-6,Verified
52-7,Verified
53-0,Verified
55-0,Crashed
55-2,Crashed
55-3,Verified
55-4,Verified
55-5,Verified
55-6,Verified
55-7,Verified
56-0,CFG not generated
```

LISTING 6.5 – The Verification Results For Property 2 Using FLIP Fault Model

Understanding the results

After obtaining the results, it is important to understand why the simulation of a specific fault model created a vulnerability.

In the PRESENT algorithm, the *Property 2* was violated only two times by fault models JMP and JBE. Listing 6.6, shows the results of the three defined properties on PRESENT obtained by applying the JBE fault model. Each line gives three information:

— the mutant binary number where the vulnerability was detected.

— the fault model that was applied.

— the property that was checked.

```
1  324,F5,P1
2  512,F5,P1
3  713,F5,P1
4  762,F5,P1
5  828,F5,P1
6  929,F5,P1
7  1117,F5,P1
8  1136,F5,P1
9  1136,F5,P2
10 713,F5,P3
```

LISTING 6.6 – The Results For Property 2 Using JBE Fault Model

Observe that for *Property 2*, only one vulnerability was detected (see line 9 in Listing 6.6). From the experimental result information, it is known that the fault injection targeted the conditional instruction at byte address `0x043c`. This corresponds to the opcode `0F86DAFCFFFF`, which corresponds to the instruction `JBE 0xFFFFFCE`.

In order to understand the correspondence between this instruction and the source code, we generate a listing file which is a mixed source and assembly list using GCC. The generated file combines the source code and the assembly code, this file is generally used in debugging programs.

Listing 6.7 presents the part of the listing file that shows the `JBE` instruction. Notice that in line 14, `JBE .L24` corresponds to the instruction `JBE 0xFFFFFCE`. This is the conditional jump related to the `while` loop in line 47 Listing 6.1.

In assembly the while loops are as follows:

— enter the loop,

— jump to the end of the loop,

— check the condition,

— if the condition is true, jump back to the start of the loop,

— if the conduit is false, exit the loop.

In this case the conditional sum corresponds to the jump at the end of the while loop which normally will jump back to the start of the loop since the condition is satisfied. But after the simulation of the fault injection the conditional instruction will change from `JBE 0xFFFFFCE` to `JBE 0x39`. This modification in the jump address will branch to the end of the program instead of branching back to the start of the while loop. In this case the value stored in the variable `state[]` remains unchanged, and corresponds to the initial value which is the plaintext. So instead of outputting the ciphertext the program outputs the plaintext.

```
1
2  156:PresentAssert.c **** //  ****************** End Key Scheduling ******************
3  157:PresentAssert.c ****      round++;
4  389                    .loc 1 157 0 is_stmt 1
5  390 042b 0FB64424      movzbl  20(%esp), %eax
6  390      14
7  391 0430 83C001        addl  $1, %eax
8  392 0433 88442414      movb  %al, 20(%esp)
9  393                  .L4:
10  80:PresentAssert.c **** //do
11 394                    .loc 1 80 0 discriminator 1
12 395 0437 807C2414      cmpb  $30, 20(%esp)
13 395      1E
14 396 043c 0F86DAFC      jbe .L24
15 396      FFFF
16 158:PresentAssert.c ****  }
17 159:PresentAssert.c ****  //while(round<31);
18 160:PresentAssert.c **** //  ****************** addRoundkey ************************
```

LISTING 6.7 – Listing File

Chapter 7

Discusion and Limitations

This chapter concludes the first part of this thesis. This chapter first discusses the FIVD process and its extension presented in section 4.1. It discusses the results of the experiments presented in sections 5.1 and 5.2. Then, it discusses the limitation of the proposed process and its implementation, and it presents possible solution that will be implemented in the second part of this thesis.

7.1 Discusion

Fault injection has recently been increasingly used to test system robustness. One of the main objective of this thesis is to propose a new approach of evaluating the robustness of systems against fault injection attacks.

In part I, the focus was on the software-based fault injection approach. A formal process that uses model checking to detect fault injection vulnerabilities in binaries was presented. This process supports the detection of many varieties of fault injection vulnerabilities, and does not rely on any particular system architecture, fault model, or other restricted choices (as are common in the literature).

The FIVD process was designed as a solution to asses the robustness of systems against fault injection attacks, with respect to the following characteristics (as presented in section 3.2):

— Automation
— Binary level
— Fault model
— Architecture independent
— Formal method

First, to have an automated process that will facilitate the task for future users. The FIVD process satisfies this point since it is fully automated. The user only needs to provide as an input the program within the properties to verify. The process script then runs the following steps automatically: circle validates the properties on the given program; circle simulates the fault injection using variety of fault models; circle if applied, run the pre-analysed step to eliminates invalide mutants; circle checks if the generated/selected mutants satisfies the properties; circle processes the results and provides the list of mutants that represent vulnerabilities.

Second, to have a process that operates at the binary level. In comparaison to [68, 35, 110, 140], the FIVD process simulates the fault injection directly at the binary level. Fault injection simulation at the binary level is considered to be more representative

of faults that can occur in physical fault injection, it also allows a fine granularity since it is possible to consider bit fault model. But in Part I of this thesis, the fact that the properties are specified at the source code level, then compiled to produce the program binary within the properties, is a problem. Since it is possible that the fault injection simulation modifies the verified properties, or modifies the program in a way that the verification results is no more valid. This problem will be addressed in part II.

Third, to have a process that will not be limited to a certain kind of fault model. The FIVD process is not limited to specific fault model. Contrary to [145] a bit flip fault model is possible using the FIVD process. The FIVD process is not limited to control flow fault models unlike [110]. To simulate the injection of faults, FIVD process uses the SIMFI tool, that supports a variety of fault models (see Section 4.3). The SIMFI tool can also be extended to other fault models with minimum developing efforts.

Fourth, to have an architecture independent process. Contrary to [47], the FIVD process is not based on a fixed hardware model. The FIVD process is a simulation-based approached, it is not implemented on specific materiel as in [118]. The FIVD process was not developed to asses the robustness of a particular system within a specific architecture, thus the process does not rely at any materiel parameters. But in the implementation of the process the instruction set of the program is taken into consideration. The toolchain used in the implementation of the FIVD needs to support the corresponding architecture.

Fifth, to provide reliable results. The use of model checking gives a formal proof for the detected vulnerability, which gives a guarantee to the obtained results. The fact that the program is modelled allows the exploring of all the paths while checking the properties, instead of considering a single path of execution. The use of model checking comes with drawbacks as well. Due to the state space explosion problem, the verified program needs to respect a limited size. In the first part, bounded model checking is used to address partially this problem, but those not resolve infinite loop cases for example.

The originality of the FIVD process lies in succeeding to combine the five characteristics presented above. Compared to previous works [35, 47, 68, 97, 107, 110, 118, 140, 145] the FIVD process tick all the boxes (see Table **??**).

Part I presented in detail the proposed FIVD process, but also an extension for larger programs that correspond to real applications implementations. The process was then applied to a small example as proof of concept. The extended version was proposed to gain time when applying the process to real cryptographic algorithm applications.

Experimental results demonstrate the efficacy of the process by testing a variety of fault models on the motivating example and the cryptographic algorithms (PRESENT and SPECK).

The results of applying this process yielded 82 vulnerabilities, 73 in PRESENT and 9 in SPECK. Both PRESENT (9 vulnerabilities) and SPECK (4 vulnerabilities) were vulnerable to faulting jump instructions that allowed encryption to be entirely bypassed. For SPECK a further 5 vulnerabilities exist, 2 by inserting non-operation byte values, and 3 by flipping individual bits. PRESENT was also found to be vulnerable to 64 different CA vulnerabilities, mostly through bit flips, but also through nopping instructions and jump target modifications. These were all found towards the end of the binary, where the last round of encryption occurs. SPECK was not found to be vulnerable to the CA vulnerabilities tested here. This is mostly due to

the number of rounds (21) to achieve encryption and that 21 is not one bit flip away from either 9 or 10. These results indicate that some kinds of attacks may be more or less achievable depending on the structure of the code used, and so some care should be taken when choosing how to implement a fault resistant binary.

7.2 Limitations

This section discusses the limitation of the FIVD process implementation. There are several limitations with the implementation chosen here. Indeed the implementation used was merely the easiest to combine effectively to implement the process.

The choice to use MC-Sema was to be able to work with LLVM-IR. The choice of LLVM-IR is due to the fact that it is being a widely used intermediate representation language that is supported by many tools. However, there are limitations with MC-Sema that may limit future work. MC-Sema supports only (some of) the instructions of X86 architecture [136] and so in the second part of this thesis MC-Sema will not be used and other architecture (ARM) will be supported.

The LLBMC model checker is sufficient for the safety properties but does not support liveness properties. Thus although LLBMC was sufficient for the proof of concept in this part, Part II will exploit a non-bounded model checker that can also accept liveness properties. In particular, a model checker that can produce traces (LLBMC can, but not combined with MC-Sema) would aid in understanding vulnerabilities and analysing results.

Fault injection was implemented with SimFI tool for this thesis, although several tools already exist to simulate fault injection attacks on software [67, 71]. However, these tools are limited by various choices that make them unsuitable for the process here (hence their lack of use in the implementation). Several are only able to inject faults into intermediate representations, and not into executable binaries [110, 135, 142], thus being unable to simulate faults that appear only at the executable binary level. Others have different limitations, such as: specific hardware platforms [107, 120], specific source code languages [35, 89, 142], or requiring simulating drivers [37]. Despite these limitations, many include useful techniques or developments that could be incorporated into future development of a general fault injection tool for executable binaries.

The pre-analyse step in the extended version of the FIVD process, was used to eliminate mutants that fail to execute or to output a result, in order to gain time using model checking. In a general setting, this approach could be considered as ?naive?, since it considers a single trace of the executable binary. This restriction may lead to fault negatives, i.e. vulnerabilities detected as crashes, etc., instead of vulnerabilities in respect to the defined properties. Note that, in the context of use cases considered in this thesis, this limitation has not been observed, as in the original implementation branches do not depend on input data. A comparaison of the results of the PRESENT algorithm was done with the FIVD process without the pre-analyse step, which detected the same number of vulnerabilities.

In a more general setting, techniques relying, for example, on pruning [100, 125] may be considered instead for future work. Note that, the pre-analyse step was not considered in Part II as we relied on statistical model checking techniques.

Complementary research is to explore ways to inject faults intelligently. This could exploit knowledge of the property to inject faults that would lead to property violations, yielding improved efficiency of experiments.

Regarding properties, another area of future work is to consider how to extract properties automatically from the binary (or source code). There is some existing work in this area [122, 153] although they focus upon high level behaviour rather than binary code.

Currently the process identifies vulnerabilities, but does not suggest fixes or countermeasures. Automatically generating countermeasures is non-trivial, although if countermeasures to particular faults are known future work could suggest or implement them automatically. Perhaps more significantly, these countermeasures could be checked immediately using the process here and so their effectiveness verified immediately.

Part II

Bridging Software-Based and Hardware-Based Fault injection Attacks

Chapter 8

Overview of Part II

This chapter introduces the second part of this thesis. This part explores the combination of software-based and hardware-based fault injection approches. This chapter first gives a general introduction to the second part of this thesis, by exploring avantages and disadvantages of both approaches, and by explaining why a potential combination of the two approaches will be beneficial. Next it recalls existing work that intended to combine the two approach, and explains the novelty of our contribution compared to others. Finally, it presents the two case studies used to demonstrate our claims. The two presented case studies will be used in the experiments presented after in chapter 10.

Sommaire

8.1 Introduction

There are two main approaches to detect fault injection vulnerabilities: software-based and hardware-based approaches. Software-based approaches simulate fault injection on some aspect of the program and detect whether some properties of the program have been altered to yield a vulnerability [31, 55, 107, 110]. Hardware-based approaches use direct experimentation on the physical hardware while the program is being executed. In this approach vulnerabilities are detected by observing the hardware during the experiment and analysing the output after execution [10, 14, 114, 130].

In part I the research of this thesis was directed to focus on the software-based technique only. An automated formal process to detect fault injection vulnerabilities on binary was proposed. The experiment results showed the efficiency on the proposed approach by detecting vulnerabilities on a toy example but also real cryptographic implementations.

The software-based process results in an exhaustive list of all possible theoretical vulnerabilities given specific fault models. In order to validate these found vulnerabilities in practice, the application of a hardware-based fault injection is needed. Notably, the combination of software-based and hardware-based approach, will strengthen both side. On the one hand, the software-based approach can be fined-tuned in order to take properties of the hardware fault vulnerability into account. On the other hand, using the knowledge given from the software-based approach may help to

understand and indicate the underlying fault models given the output of the experimental hardware-based approach.

Software-based approaches can either be based on simulations or implementations, both directions have been studied in the published research so far. In this thesis, the focus was on the simulation-based approach using formal verification techniques to detect fault injection vulnerabilities. A new automated formal approach was proposed and validated in Part I.

For the hardware-based approaches there exists a variety of techniques as well that will be presented later in section 8.2. In this thesis EMP techniques were used to perform hardware experiments.

Both of the software-based and hardware-based approaches have advantages and disadvantages as detailed in Sections 2.1.1 and 2.1.2.

The correspondence between software-based and hardware-based fault injection vulnerability detection has not been widely explored yet. To the best of our knowledge only one work attempted to combine the software and hardware approaches (see section 8.2). In the eighties [39] used a very limited technique focusing on the fault detection time. The experiments concluded that the results of the two approaches do not map. The research focused only on the fault detection time.

This thesis sets out to remedy this and to bridge the gap between software-based and hardware-based fault injection vulnerability detection. This is achieved here by performing both software-based (i.e. simulation) and hardware-based (i.e. EMP on hardware) fault injection vulnerability detection on two case studies. The results of the two approaches are compared to explore how closely the two kinds of approaches coincide. The results of these experiments yielded several interesting outcomes:

— The software-based approach is able to find genuine fault injection vulnerabilities. However, there are also false-positive results where the software-based approach claims a vulnerability exists that was not feasible to reproduce using the (EMP) hardware-based approach. This indicated that although software-based approaches may be useful in identifying *potential* fault injection vulnerabilities, not all such vulnerabilities are validated in practice.

— The hardware-based approach did *not* match to any single fault model of the software-based approach. By having comprehensive results from the software-based simulation it was possible to determine that the (EMP) hardware-based approach did not have a consistent or exact effect on the hardware. Although this is not surprising (specially since EMP is inexact at best), this indicates that simulations that consider only a single fault model may not correspond well to EMP (or other kinds of hardware-based attacks).

— The two approaches coincide. That is, both approaches agree on the effect and the location of fault injection vulnerabilities (and other behaviours). The results here indicated that although the software-based approach had false-positives, there were no false-negative results when considering the fault models used here. **This indicates that software-based detection can indicate likely locations for vulnerabilities, and hardware-based approaches can be used to confirm (or refute) their feasibility**.

Combining both approaches can be used to rapidly locate genuine fault injection vulnerabilities, even in code without known weaknesses. This thesis presents a

method to use the software-based approach to identify the most potentially vulnerable locations and then (with some calculation) these can be tested and confirmed (or disputed) using hardware-based approaches. In practice this combined approach can vastly reduce the number of hardware experiments required to demonstrate a vulnerability; here reducing the number of experiments from tens or hundreds of thousands to just 210. Further, when applied to code without known weaknesses this can be used to rapidly determine if vulnerabilities exist.

8.2 State of the Art

This section recalls related works that use combined software-based and hardware-based approaches for detection of fault injection vulnerabilities.

The only attempt to combine software-based and hardware-based approaches was in 1987, where Czeck. et al. [39] compared the results of software simulation fault injection and hardware experiment fault injection with the objective to test the correctness and dependability of system. At this time the hardware fault injection was limited to the pin level modification. The software simulation fault injection was done by modifying program data or control. The authors concluded that although the software-inserted faults do not map directly to hardware-inserted faults, experiments show software-implemented fault insertion is capable of emulating hardware fault insertion, with greater ease and automation. Compared to this work, the experiments conducted in [39] were limited, the objective was to test the system using different approaches and no combination proposition was given at the end of the experiments.

Other works [3, 119] propose exploiting the hardware materiel by simulation fault injection on the target hardware. This technique is the software-implemented approach which is considered as a software-based approach and not hardware-based one.

In [119] the authors propose combining the Lazart process with the Embedded Fault Simulator (EFS) [20]. This extends from the capabilities of Lazart alone by adding lower level fault injection analysis that is also embedded in the chip with the program. The simulation of the fault is performed in the hardware, using EFS. However, EFS is a software-implemented based approach, it can not be considered as a hardware-based approach. The combination in this work could not be compared to the work presented in this thesis.

Similarly in [3] where experiments are used for testing the TTP/C protocol in the presence of faults. Rather than attempting to find fault injection attacks, they injected faults to test robustness of the protocol and simulation of hardware faults. They combined both software-implemented approach on the hardware and simulation-based approach on the software, comparing the results as validation of the proposed TTP/C protocol model.

The research conducted during this thesis did not find other works than [39] presented above which tried to combine the software and hardware fault injection approach. The majority of existing works focus either on software-based approaches or hardware-based approaches. The combination of the two approaches is not enough exploited yet.

8.3 Case Studies: Control flow Hijacking and Backdoor Attack

This section presents the two case studies used to exploit the connection between the software-based and hardware-based fault injection approaches.

The first case study is the control flow hijacking attack. In a control flow hijacking attack, the attacker objective is to take over the target system by hijacking its normal control flow. The control flow hijacking attack is known to be the general class of attacks such as buffer overflow, integer overflow attacks, format string vulnerabilities, and backdoor attack. Control-flow hijacking attacks have become a serious problem [56, 103] due to the ease of exploring the created vulnerabilities and get interesting outcomes.

The second case study is a backdoor attack. A backdoor attack is a sub classe of the control flow hijacking attack, where the attackers objective is to succeed redirecting the control flow to his own injected code, or his choice of existing code which constitutes the backdoor. The backdoor is considered to be an undocumented functionality in the system (program) that could be implemented by malicious insiders at the factory house, or added during the manufacturing for testing and debug purpose. Previous research [131] showed the presence of deliberately inserted backdoor in chip used in both military and sensitive industrial applications. Other research [82, 104] showed that, using fault injection, it is possible to activate backdoor code.

The rest of this section presents in details the chosen case studies by showing their source code and assembly code, and explaining the objective of the attacks, and how the vulnerabilities are detected.

Note that the case studies contain *trigger* instructions that change the voltage of some pins observable to the hardware fault injection tools. These were used to improve precision in the hardware calibration as described in Section 9.3. However, no result in this work relies upon the existence of these triggers.

Control Flow Hijacking

Control flow hijacking case study [30] is chosen to have a known class of vulnerability that is straightforward to understand. The example here is presented in the original C source code, and in the assembly instructions for ARMv7-M. The source code of the case study programs is available in the git repository [1]. Finally, for the case study the correct, vulnerable, and incorrect program executions are defined.

This case study is chosen to demonstrate a control flow hijacking vulnerability. The goal for the attacker is to output a specific value (`0x55555555`) that can only be reached by hijacking the control flow of the program execution.

The `test_persistence` function that is of interest is shown below.

```
uint32_t test_persistence (void){
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7, GPIO_PIN_SET);
 uint32_t status = 0;
 if (pin_correct==1) {
   status=0xFFFFFFFF;
 } else {
```

1. https://gitlab.inria.fr/rlasherm/ARMv7M-under-attacks.

```
    status=0x55555555;
  }
  HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7, GPIO_PIN_RESET);
  return status;
}
```

LISTING 8.1 – Control Flow Hijacking Case Study C Code

Here the attacker wishes to hijack the function control flow to return `0x55555555` even when `pin_correct` has the value `1`. Since in the code being experimented `pin_correct` always has value `1`, the program behaviour can be defined to be one of the following outcomes. The *correct* behaviour for this case study is to return `0xFFFFFFFF`. The program is *vulnerable* when the return value is `0x55555555` (achieved via some form of fault injection). Any other return value is considered to be *incorrect* program execution. Note that if the program does not terminate or provide a return value due to the fault injection, it is considered to be *crashed*.

The corresponding ARM-v7 assembly instructions for the `test_persistence` function are shown below.

```
08000aa0 <test_persistence>:
 8000aa0: b510 push {r4, lr}
 8000aa2: 480a ldr  r0, [pc, #40]
 8000aa4: 2180 movs r1, #128
 8000aa6: 2201 movs r2, #1
 8000aa8: f001 f91c bl 8001ce4 <HAL_GPIO_WritePin>
 8000aac: 4b08 ldr  r3, [pc, #32]
 8000aae: 4807 ldr  r0, [pc, #28]
 8000ab0: 681b ldr  r3, [r3, #0]
 8000ab2: 2180 movs r1, #128
 8000ab4: 2b01 cmp  r3, #1
 8000ab6: bf0c ite  eq
 8000ab8: f04f 34ff moveq.w r4, #4294967295; 0xffffffff
 8000abc: f04f 3455 movne.w r4, #1431655765; 0x55555555
 8000ac0: 2200 movs r2, #0
 8000ac2: f001 f90f bl 8001ce4 <HAL_GPIO_WritePin>
 8000ac6: 4620 mov  r0, r4
 8000ac8: bd10 pop {r4, pc}
 8000aca: bf00 nop
 8000acc: 40011000 .word 0x40011000
 8000ad0: 2000001c .word 0x2000001c
```

LISTING 8.2 – Control Flow Hijacking Case Study Assembly Code

There are several instructions that are of significance to correct program execution.

- The instruction at `8000ab0` that loads to `r3` the value at memory address `[r3, #0]`.
- The instruction at `8000ab4` that compares the register `r3` with the value `#1`.
- Then, the instruction at `8000ab8` loads the value `#4294967295(=0xFFFFFFFF)` into the register `r4` if the prior condition is satisfied.
- Similarly, the instruction at `8000abc` loads the value `#1431655765(=0x55555555)` into `r4` when the prior condition is not satisfied.
- Then, the instruction at `8000ac6` that moves to the return register `r0` the return value from register `r4`.

Observe that faulting any of these would certainly have an effect on correct program execution, since instead of returning `status=0xFFFFFFFF` the program will return either `status=0x55555555` or another value. Although, this does not mean that faults in other instructions cannot also cause changes to program execution.

Backdoor Attack

This section recalls the Fault Activated Backdoor program from [30]. The core of the weakness in the code is a `backdoor` function (shown in Listing 8.3) that is hidden in the program but cannot be reached by any execution path. The normal behaviour of the program includes encryption with AES [132] yielding a ciphertext. The `backdoor` function (when executed) replaces the ciphertext with the AES key, thus allowing an attacker to observe the "ciphertext" and in practice to learn the key. However, under normal conditions the `backdoor` function can never be executed, and so should not be detected by static or dynamic code analysis.

The weakness here is built into the code in the `blink_wait` function shown in Listing 8.3. The value of `wait_for` is defined to be `3758874636`, which has two special properties. Firstly, this value is too large to be loaded within a single ARM-v7 instruction and so the value is stored as a seperate word in the assembly code. Secondly, this value if interpreted as an instruction corresponds to a jump to a specific location (in practice the location of the `backdoor` function).

```
void blink_wait()
{
  unsigned int wait_for=3758874636;
  unsigned int counter;
  for(counter=0;counter<wait_for;counter+=8000000);
}
void backdoor(void)
{
  int i;
  for(i = 0; i < DATA_SIZE; i++)
  {
    ciphertext[i] = key[i];
  }
  HAL_GPIO_WritePin(LED3_GPIO_PORT, LED3_PIN, GPIO_PIN_SET)
     ;
}
```

LISTING 8.3 – Backdoor Case Study C code

The corresponding assembly code for the `blink_wait` function is shown in Listing 8.4. Observe that the value of `wait_for` is stored at the end of the function at address `80005cc` immediately after the `POP` instruction at address `80005ca`. Thus, an attacker that can cause this `POP` instruction to be skipped or interpreted as something else (e.g. a `MOV`, `ADD` or `LDR` as observed in Section 10.1) would then execute this value as a jump to the `backdoor` function.

```
08000598 <blink_wait>:
8000598: b580 push {r7, lr}
800059a: b082 sub sp, #8
800059c: af00  add r7, sp, #0
800059e: 4b0b ldr r3, [pc, #44]
```

```
80005a0: 603b st r3, [r7, #0]
80005a2: 2300 movs r3, #0
80005a4: 607b str r3, [r7, #4]
80005a6: e005 b.n 80005b4 <blink_wait+0x1c>
80005a8: 687b ldr r3, [r7, #4]
80005aa: f503 03f4  add.w r3, r3, #7995392 ; 0x7a0000
80005ae: f503 5390 add.w r3, r3, #4608 ; 0x1200
80005b2: 607b str r3, [r7, #4]
80005b4: 687a ldr r2, [r7, #4]
80005b6: 683b ldr r3, [r7, #0]
80005b8: 429a cmp r2, r3
80005ba: d3f5  bcc.n 80005a8 <blink_wait+0x10>
80005bc: f7ff ffe2 bl 8000584 <fast_trig_up>
80005c0: 2003 movs r0, #3
80005c2: f000 f8af bl 8000724 <wait>
80005c6: 3708 adds      r7, #8
80005c8: 46bd mov sp, r7
80005ca: bd80 pop {r7, pc}
80005cc: e00be00c      .word 0xe00be00c
```

LISTING 8.4 – Backdoor Case Study Assembly

The *correct* behaviour of the program is to output a normal ciphertext. The *vulnerable* behaviour is to output the encryption key, or to ever execute any code inside the `backdoor` function. (This choice was to detect potential vulnerabilities that could be exploited regardless of whether the key was completely leaked.) *Incorrect* was any other output, and *crashes* were failure to terminate or provide output as before.

Chapter 9

Process, Implementation and Methodology

This chapter presents the processes adopted to conduct experiments using the two approaches, the software-based and hardware-based fault injection approaches. This chapter is divided into three sections: in section 9.1 the software-based and hardware-based process are presented in details. The improvement done on the FIVD process are highlighted. In section 9.2 the implementation of the processes presented in the previous section are detailed. Similarly to the software-based process, in the software-based implementation the improvement are highlighted. Lastly in section 9.3 the methodologies followed to conducted the experiments are presented.

Sommaire

9.1 Software and Hardware based Process

This section presents the software-based and hardware-based process used to conduct the experiments. The software-based process presented in this section is an improvement of the FIVD process presented in Chapter 4. The hardware-based process presented in this section explain the steps to follow in order to conduct the hardware experiments.

Software-Based Process

This section recalls the key aspects of the automated formal process for detecting fault injection vulnerabilities in programs as it was presented in chapter 4. This process has been demonstrated to efficiently find fault injection vulnerabilities on cryptographic algorithms using a wide variety of fault models as shown in chapter 5. The process has some limitations as discussed in chapter 7. These limitation were mainly related to implementation constrains. This section presents an improved version of the FIVD process by addressing the limitations presented previously.

An overview of the process as depicted in Figure 9.1 is as follows. The process starts with the binary file and the file of the properties to check upon the binary. The properties are validated to hold on the model using SMC. The fault injection is then

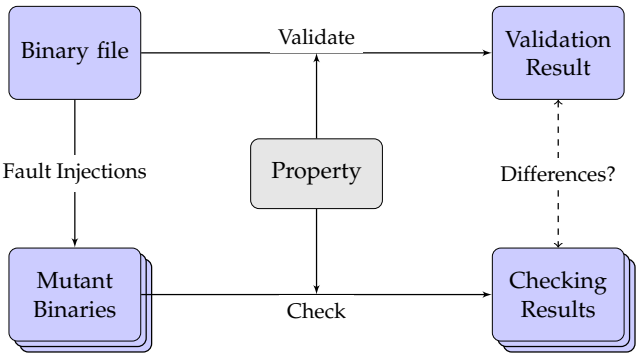

FIGURE 9.1 – Software Process Diagram

simulated on the binary file in order to produce mutant binaries. The properties are then checked upon the mutant binaries using SMC. A difference in the results of the validation and checking the property indicates a fault injection vulnerability created by the simulated fault injection and instance of the fault model.

The presented process in this section differs from the version of the process presented in section 4.1 in the following points.

First point, the properties are not added to the source code file. The properties are specified in a separate file that will be used directly by the model checker. The fact that the properties are specified in a separate file insures that there is no risk of modifying the properties when simulating the fault injection on the binary. This was not the case in the previous version, where the properties were specified in the source code and then the source code within the properties were compiled to generate the executable binary on which the fault injection were simulated. So there were chances that the properties can be modified by the simulation of the fault injection which is no more the case with this improvement.

Other advantage related to the properties, is that now the properties are specified at a lower level, where it is possible to express the properties using the *register*, *program counter*, etc. This enhancement allows to insert fine-grained properties that are closer to binary level instead of the source level.

Second point, the preparation step is no more needed, the process starts directly from the binary file and not the source code. In the previous version of the process the source code was needed in order to detect vulnerabilities, it was not possible to work only with the binary file. Now that the process starts directly from the binary it is possible to test the robustness of programs without having access to their source code. But it is important to mention that it might not be feasible to specify properties only with the binary file. In this work the properties express the behaviour the program should (or shouldn't) respect. If the source code is not available it might be hard to understand the program and so hard to specify the properties to verify.

The process in this section is similar to the main steps of the process in Section 4.1 in the main steps. The first step of the process still is the validation step which consists of validating that the original binary satisfies the property before going to the next step of the process. The second step of the process still is the simulation of the fault injection on the binary file to produce mutant files, which corresponds to the binary file after injecting the specified fault using a fault model. The third step of the process still is checking the property on the generated mutant files after the fault injection simulation. The fourth step of the process is still comparing the validation and the checking results to detect vulnerabilities.

Hardware-Based Process

This section overviews the hardware process used for detecting fault injection vulnerabilities in programs. This process is common to many prior works in this domain [30, 51, 109, 120].

An overview of the process is as follows. Preparation step of the hardware process is to choose the hardware materiel to induct the fault, based on the target hardware and the desired effect.

1. The first step is to experiment on the chosen target hardware with the chosen hardware fault induction technique. This step consists of exploring the spacial surface of the target hardware while manipulating the parameters to induce the fault using the induction technique. This step concludes when a configuration is found that allows the hardware fault injection to change program execution. This step requires a lot of time and expertise, since exploring all the possible combinations of spacial, temporal, and induction parameters is not feasible in reasonable time.
2. The second step is to analyse the target program. This step consists of detecting potential known vulnerabilities in the program.
3. The third step is then to load a program onto the target hardware that has a believed vulnerable point. For the hardware-based approach, contrary to the software-based approach, the objective of the experiment is to show that a predicted vulnerability exists. This is the reason why the loaded program is known to be vulnerable before conducting the experiments.
4. The fourth step is to try and align the injected fault with the believed vulnerable point to demonstrate a fault injection vulnerability. The objective of this step is to align the parameters set at the first step with the known vulnerability loaded in the second step. Similarly to the first step, parameter manipulation is needed to try and detect the known vulnerability in the loaded program by injecting fault using the chosen hardware induction technique.

A vulnerability has been demonstrated if the fault injection can change the program execution in the desired way with significant consistency.

9.2 Software and Hardware based Implementation

Software-Based Implementation

The software simulation experiments were performed by an implementation of the process described in Section 9.1. The process was extended to operate on ARM-v7 and to use SMC (as opposed to on X86 and using BMC respectively, see section 4.2).

The implementation of the process can be seen in Figure 9.2. The implementation begins with a binary file for ARM-v7 architecture and the properties specified in B-LTL (in separate files).

The binary is translated to *Reactive Module language* (RML) using the *ARM to RML* (ARML) tool. ARML is a translation tool that has been developed during this thesis to translate from ARM-v7 binaries to RML models. RML [83] is a state-based language based on the Reactive Modules formalism [4] and used as the input language for Plasma Lab [86].

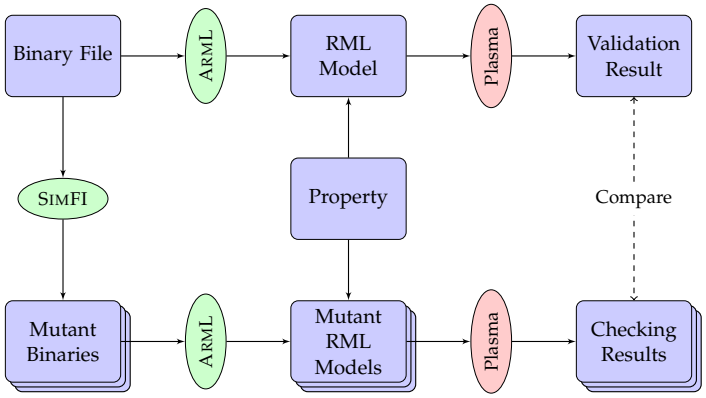

FIGURE 9.2 – Software Implementation Diagram

The specified property is then validated to hold on the generated RML model using the SMC Plasma Lab [86]. Both the RML model file and the property file are given as input to the SMC Plasma Lab tool, the algorithm to do the verification and the number of simulations are then selected in the Plasma Lab tool. The process is fully automated, such that a dedicated script launches the checking step and receives the results.

The mutant binaries corresponding to simulated fault injections are generated using *Simulation for Fault Injection* (SIMFI) tool. The SIMFI tool is a tool that has been developed during this thesis, it simulates a wide variety of fault injection attacks on binaries. The tool takes a binary as an input (regardless of the binary's architecture). Based on the chosen fault model a mutant binary is generated, representing the simulation of the chosen fault injection attack.

The RML models for the mutant binaries are generated using the ARML tool. The properties are then checked on the mutant models using SMC with Plasma Lab. Finally the results of model checking the mutant models and the binary file model are compared for statistically significant differences. Due to using SMC, minor differences can occur because of the statistical model checking.

Hardware-Based Implementation

The hardware process also follows the standard approach to hardware-based fault injection technique as presented in Section 9.1.

The chosen target hardware is a STM32 Value-line discovery board with ARM Cortex-M3 core micro controller running at 24MHz. The STM32 Value-line discovery board is a low cost STM32 Value-line, that is widely used by experts but also by beginners. This board has all the needed features to explore and evaluate STM32F100 microcontrollers.

The chosen fault injection induction method is to induce a fault via EMP. The EMP signal is initiated by a KEYSIGHT 33509B Waveform Generator that sends a signal through a KEYSIGHT 81160A Pulse Function Arbitrary Noise Generator (a high precision pulse generator that helps in the manipulation of the signal). The signal is then amplified using a MILMEGA 80RF1000-175 RF AMPLIFIER. Finally, the signal is sent to a Probe RF B 0.3-3.

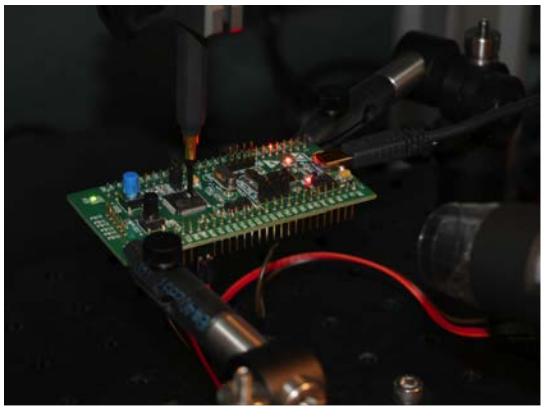

FIGURE 9.3 – Hardware Implementation Probe Location

Initial experiments were then conducted to find a configuration that allowed consistent program execution disruption. In practice, this was achieved by placing the probe above the chip as depicted in Figure 9.3.

Further experiments were conducted to calculate the latency of the various components. This allowed calculation of the timing between the injection of the fault and observing the effect. Further, this allowed calibration of the minimum and maximum possible delay between fault injection and observations of effects. The delay between the injection of the fault and the observed effect was 0.08μs to 0.12μs.

9.3 Software and Hardware based Methodology

This section discusses the experimental methodology used to conduct the experiments in Part II of this thesis. The overall methodology is as follows.

1. The first step is to take the case study and perform extensive software simulations to identify as many potential vulnerabilities as possible. Incorrect program execution is also identified to help in later stages of the methodology. Finally, crashes and other failures of program execution are exploited for calibration as described later.
2. The second step is to perform hardware fault injections on the entire function and to identify which configurations yield statistically significant changes in program execution.
3. The third step is to compare the software and hardware results to:
   — identify achievable fault injection vulnerabilities using the hardware results;
   — identify likely hardware fault models using the software results;
   — and to demonstrate that software-based and hardware-based fault injection techniques coincide.

The rest of this section details the environment and implementation for the experiments.

Software-Based Methodology

For software simulation various fault models can be simulated by SIMFI. Since the EMP used here (see Section 9.3 below) does not have a single consistent fault model [96], multiple fault models were considered here. The fault models tested here are as follows.

Z1B The *zero one byte* fault model (Z1B) simulates setting a single byte to zero (regardless of prior value). This fault model corresponds to a malicious attack that is commonly achievable attack in practice [123, 137].

Z1W The *zero four bytes* fault model (Z1W) represents setting four bytes to zero (again regardless of prior value). This is similar in concept to the Z1B fault model and attack, but captures behaviour more related to the hardware model, since it reflects faulting some piece of the hardware that operates on words rather than bits or bytes (such as the ARM Cortex-M3 bus used here) [38].

NOP The *ARM NOP* fault model (NOP) sets the targeted operation to a non-operation (NOP) instruction for the chosen architecture (in this case `0x00BF` for ARM-v7). The concept behind this model is that it simulates skipping an instruction, a common effect of many runtime faults [97].

TAM The *tamper* fault model (TAM) sets a byte to a specific value. Here the TAM fault model sets the value of byte to `0xFF`. This is opposite concept to Z1B fault model and attack, this may be an effect of EMP. The choice of using this here is to consider when an EMP may fault the chip with the opposite electromagnetic effect (i.e. set all bits to 1's instead of 0's). The TAM fault model can be explained by the stuck-at fault. This fault results in a line of a logic circuit being permanently stuck at a logic one [2].

FLIP The *flip* fault model (FLIP) simulates the flipping of a single bit, either from 0 to 1 or from 1 to 0. This fault model is highly representative of many kinds of faults that can be induced, ranging from those due to atmospheric radiation, to software effects such as the rowhammer attack [76].

Note again, that the SIMFI tool is not restricted to single faults, but the fault models considered here chosen to fit to the outcome of the hardware-based approach.

The software simulation experiments were performed to simulate all listed fault models on all possible addresses within the target function of the case study. The outcomes were then classified as: correct, vulnerable, incorrect, or crashed.

The simulations were conducted on a virtual machine configured with one CPU, 11.7GB of RAM, and 179.4GB of disk space running Linux Ubuntu 16.04 LTS. The virtual machine was hosted on a Macbook Pro with 3.1 GHz Intel Core i7 processor, 16 GB of RAM, and running macOS High Sierra 10.13.3.

Hardware-Based Methodology

For the case study, the program was loaded onto the target hardware. The triggers were then used to calibrate the fault injection hardware tools and to verify the latency calculations were correct. Further, the minimum and maximum clock cycle [1] count was calculated for the case study functions (using the Cortex-M3 technical reference manual [38]). These were then used to find the earliest start point and latest

1. Each clock cycle is approximately 40ns.

end point of execution of the functions being considered (including an error margin to ensure complete coverage).

Once the bounds of the execution have been calculated, hardware faults were injected at 4ns intervals starting from the earliest possible start point to the latest possible end point. The results for each execution and fault injection are then recorded. This is then repeated a large number of times to gain statistical information on the effects at each timing points. (This last step is done to account for minor inconsistencies in effects, and due to the general imprecision of EMP faults, as well as due to fault injection vulnerabilities not being achievable with high reliability in practice.)

Bridging Software-Based And Hardware-Based Approaches

This section shows how to bridge the software-based and hardware-based approaches (Sections 9.3 & 9.3 above) and then compare the results.

This comparison was done for each fault model from the software experiments with the results from the hardware experiments. The number of clock cycles were calculated (up to the fault injection point, since after this the results may be perpetuated), and then used to cross-reference with the address of the fault from the software experiments. Then, the alignment of the clock cycles were varied to see if there was a strong transition point where the hardware clearly changed from one instruction to another (since the clock cycles are not perfectly aligned, and the hardware experiments tested, injected many faults at different times within each clock cycle's length).

The above comparison was also performed for combinations of fault models, and for subsets of fault models. Each combination of fault models was compared to see if multiple fault models combined matched well with the hardware experiments. Similarly, subsets of the results within fault models were used for some fault models. The Z1B and NOP in particular were tested with subsets of their results that considered only being applied to: every second byte (i.e. at the start or end of many instructions), to every fourth byte (i.e. at the start or end of many words), and to the first or second byte of every instruction (i.e. which can be two or four bytes since the instruction lengths vary).

Chapter 10

Experimental Results

This chapter presents the experimental results of the software-based and hardware-based fault injection approaches on the two case studies presented in section 8.3. This includes: the results of the software simulation experiments alone; the results of the hardware experiments alone; and relations between the software and hardware experiments.

Sommaire

10.1 Control Flow Hijacking

This section presents the control flow hijacking case study experimental results. Section 10.1 presents the software experimental results for the control flow hijacking case study. Section 10.1 highlights the hardware experimental results. Section 10.1 overviews the comparison of the software and hardware results. Note that the experimental methodology here is the one in Chapter 9.

Software-Based Experimental Results

This section overviews the results of the software simulation experiments. Note that the result of the experiment will be presented byte-wise. The bytes are indicated by their addresses in the assembly code listed in Listing 8.2.

An overview of results of the software simulation for the control flow hijacking case study can be seen in Figure 10.1. (The red coloured bytes ■ indicate the presence of vulnerabilities and the blue coloured bytes ■ indicate the presence of incorrect results, absence of any colour indicates correct behaviour.)

Observe that all fault models indicated some vulnerabilities between bytes `800aad` and `8000ab8`. Additionally, the FLIP fault model indicated a vulnerability earlier at byte `8000aa8`. Incorrect results were detected from byte `800aab9` to byte `8000abd` by all fault models except TAM.

All fault models indicated vulnerabilities between bytes `8000ab0` & `8000ab1`, and `8000ab4` & `8000ab5`. However, there was no consensus on where the incorrect results of execution would appear amongst all the fault models (or even only the fault models that had incorrect results).

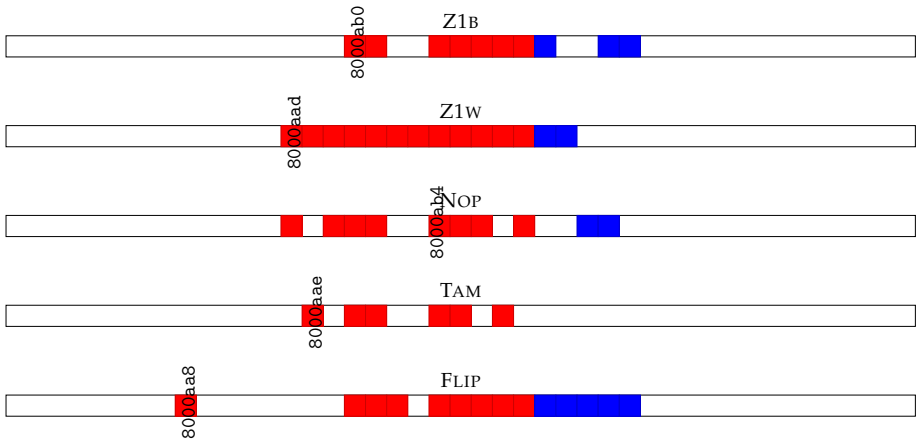

FIGURE 10.1 – Software-Based Control Flow Hijacking Results

To understand how the fault injection simulation created a vulnerability in the program, one needs to look at the binary code and the corresponding assembly instructions. The different fault model had different impact on the assembly instruction, but overall the fault injection impact on the assembly instruction can be classified into two groups of impact. The first group of impact is when the fault injection simulation changes the original instruction to an other instruction or multiple instructions.

$$InstA \rightarrow InstB$$

$$InstA \rightarrow InstB\ and\ InstC$$

The second group of impact is when the fault injection simulation does not change the original assembly instruction but modifies either the instruction address, the instruction manipulated value, or the instruction manipulated register.

$$InstA[address^1] \rightarrow InstA[address^2]$$

$$InstA[value^1] \rightarrow InstA[value^2]$$

$$InstA[register^1] \rightarrow InstA[register^2]$$

The rest of this section focuses on the instructions between bytes `8000ab0` & `8000ab1`, and `8000ab4` & `8000ab5`. In these bytes all the fault models indicated potential vulnerabilities. For each instruction, details will be given on how the simulation of the fault injection modified the byte instruction, and why this modification created a vulnerability in the program.

The instruction `ldr r3, [r3, 0]` at byte address `8000ab0` in Listing 8.2, loads the value of the variable `pin_correct` into the register `r3`. The simulation of a fault using the various fault models produces the following effects.

— Using all the fault models (except for NOP), one can change the `LDR` instruction to `MOV`, `CMP` or `STR` instruction.

— Using all the fault models, it is possible to change where the value of `pin_correct` from register `r3` to a different register (e.g. `r0`, `r7` or `r2`).

— Using the FLIP fault model, it is possible to modify the memory address from where the value will be loaded, yielding an unknown (or effectively random) value for `pin_correct`.

— Using the NOP fault model, it is possible to replace the instruction with a `NOP` instruction and so the value of `pin_correct` is implicitly set to whatever was in `r3` prior to this point in execution.

All the above effects will not set the register `r3` to the correct value of the variable `pin_correct` and so affect the comparison done later on line 11 in Listing 8.2.

The instruction `cmp r3, 1` at byte address `8000ab4` in Listing 8.2 compares the value of the register `r3` with `1`, and updates the corresponding flags of the Application Program Status Register (APSR) based on the result of the comparison. The simulation of a fault using the fault models produces the following effects.

— Using the Z1B, Z1W and FLIP fault model, it is possible to change the `CMP` instruction to `MOV`, `ADD` or `LDR` instruction.

— Using all the fault models, it is possible to change the value of the number `1` the instruction compares with the register `r3`.

— Using the NOP fault model, it is possible to replace the instruction with a `NOP` instruction.

The above fault models modification will effect the comparison of the register `r3` value with `1`, which will impact the choice of the correct branching in the following three instructions.

The instruction `ite eq` on byte address `8000ab6` in Listing 8.2 defines the APSR flags to set to be used by the following two instructions. The simulation of a fault using the fault models produces the following effects.

— Using all the fault models (except TAM), it is possible to change the `IT` instruction to `MOV`, `ADD` or `LDR` instruction.

— Using the Z1B, Z1W and NOP fault models, it is possible to replace the instruction with a `NOP` instruction.

— Using the TAM and FLIP fault model, it is possible to change the branching order the processor follows.

All of these can yield changes to the APSR flags that will in turn alter the effects of the following two instructions. In general, the alterations allow the branching behaviour to be inverted, and thus yield an effective hijack of the control flow.

The instruction `moveq.w r4, 4294967295` on byte address `8000ab8` in Listing 8.2 sets the return register `r4` to the value `4294967295` which corresponds to the value `0xffffffff`. The simulation of a fault using all fault models (except TAM) produces the following effects.

— Using the Z1B, Z1W and FLIP fault model, it is possible to change the `MOV` instruction to `ADD`, `STR` or `LDR` instruction.

— Using all except the FLIP fault model, it is possible to replace the instruction with a `NOP` instruction.

The above fault models modification will not set the return register to the correct value. So the returned value will be whatever was already in the register `r4` generally yielding an incorrect result.

Hardware-Based Experimental Results

This section overviews the results of the hardware experiments. The hardware experiments were conducted following the methodology described in Chapter 9.

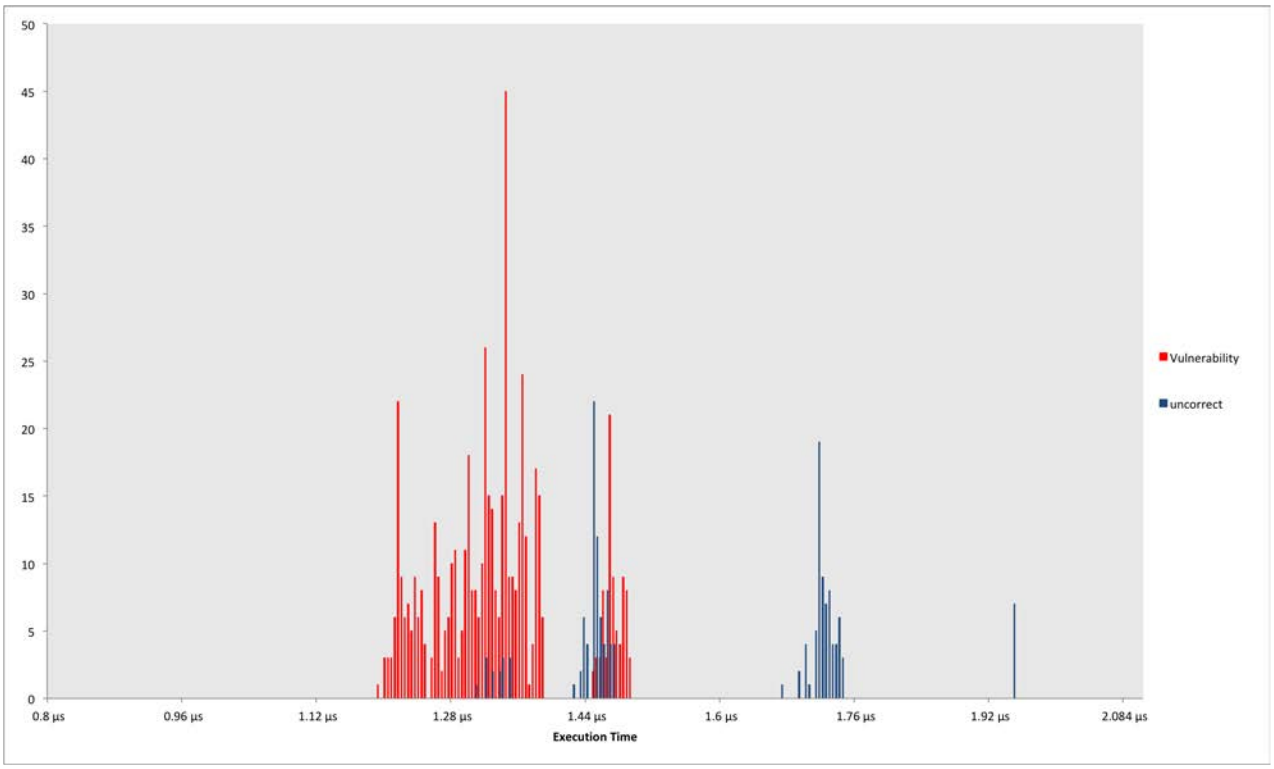

FIGURE 10.2 – Hardware-Based Control Flow Hijacking Results

Using the calculations described in Section 9.3, the earliest possible start time for the `test_persistence` was calculated to be 0.8μs, and the latest possible end time to be 2.084μs. The hardware fault injection experiments were thus conducted within this range.

An overview of the results of the hardware experiments for the control flow hijacking case study can be seen in Figure 10.2. Observe that vulnerabilities were grouped together in two groups. The larger group between 1.192μs and 1.388μs, and the smaller group from 1.448μs to 1.492μs. The incorrect results are in three groups: one from 1.308μs to 1.348μs, another from 1.424μs to 1.472μs, and a third from 1.692μs to 1.744μs. There is also a single spike of incorrect results at 1.948μs.

Combining the known timing information with the clock cycle count for each instruction (from the Cortex-M3 technical reference manual [38]), it is possible to approximate which instructions are being loaded and executed at each fault injection timing. Table 10.1 gives the range of clock cycle number needed for each instruction. In order to get the timing, a simple multiplication operation is required. Note that for this particular ARM architecture the processor fetches 32 bits at a time, which means that for a 16 bit instruction the processor will fetch 2 instruction at a time. From all the above information the hardware fault injection vulnerabilities in Fig. 10.2 can be mapped to the addresses in Listing 8.2.

— The vulnerability detected between 1.192μs and 1.232μs corresponds to the instruction at byte address `8000aac` in Listing 8.2.

— The vulnerability detected between 1.236μs and 1.312μs corresponds to the instructions at byte address `8000aae` and `8000ab0` in Listing 8.2.

— The incorrect result in 1.424μs to 1.472μs corresponds to the instructions at byte address `8000ab2` and `8000ab4` in Listing 8.2.

Assembly instruction	Clock Cycle
8000aa0: b510 push {r4, lr}	3
8000aa2: 480a ldr r0, [pc, #40]	1-2
8000aa4: 2180 movs r1, #128	1
8000aa6: 2201 movs r2, #1	1
8000aa8: f001 f91c bl 8001ce4	2-4
8000aac: 4b08 ldr r3, [pc, #32]	1-2
8000aae: 4807 ldr r0, [pc, #28]	1-2
8000ab0: 681b ldr r3, [r3, #0]	1-2
8000ab2: 2180 movs r1, #128	1
8000ab4: 2b01 cmp r3, #1	1
8000ab6: bf0c ite eq	0-1
8000ab8: f04f 34ff moveq.w r4, #4294967295	1
8000abc: f04f 3455 movne.w r4, #1431655765	1
8000ac0: 2200 movs r2, #0	1
8000ac2: f001 f90f bl 8001ce4	2-4
8000ac6: 4620 mov r0, r4	1
8000ac8: bd10 pop {r4, pc}	4-6

TABLE 10.1 – Clock Cycle Duration Per Instruction

— The vulnerability detected between 1.448μs and 1.492μs corresponds to the instructions at byte address 8000ab4 to 8000ab8 in Listing 8.2.

— The incorrect result in 1.672μs to 1.948μs corresponds to the instructions at byte address 8000ac6 and 8000ac8 in Listing 8.2.

Comparison

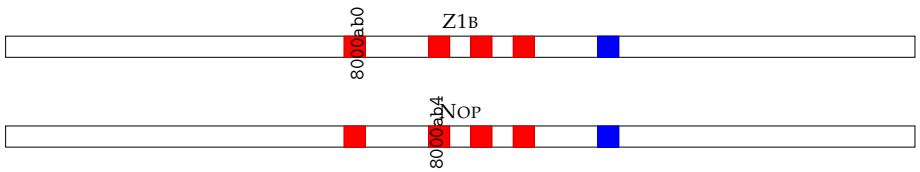

FIGURE 10.3 – Software-Based Control Flow Hijacking Results Only First Byte Instruction Results

This section compares the results of the software-based and hardware-based fault injection experiments presented in the previous two sections (10.1 & 10.1).

Overall, both approaches detected vulnerabilities in the instructions (starting) at byte addresses 8000aac, 8000aae, 8000ab0, 8000ab4, 8000ab6, and 8000ab8 in Listing 8.2. However, no fault was detected by the hardware prior to 8000aad (implying the FLIP fault injection vulnerabilities here could not be realised).

Overall both approaches detected incorrect results in the instructions (starting) at byte addresses 8000ab2 and 8000ab4 in Listing 8.2. However, the TAM fault model did not indicate any incorrect results anywhere (implying that the TAM fault model may not be accurate representations of EMP effects).

Observe that since although all the fault models detected vulnerabilities in some of the same areas as the hardware experimental results, the above implications suggest that the TAM and FLIP models do not appear to describe the effects of EMP

accurately. This leaves the setting of byte(s) to zero (Z1B and Z1W) and skipping instructions (NOP) as the best fit between the software-based results and the hardware-based results.

The Z1B fault model matches quite well with having two groups of vulnerabilities, as well as two groups of incorrect results. This corresponds closely to the hardware results that also have two distinct groups of vulnerabilities, and of incorrect results (a third less clear group of incorrect results also exists).

The Z1W fault model matches well with the vulnerable results, but also has vulnerable results that are not confirmed by the hardware. That said, the faulting of a whole word tends to produce vulnerabilities that occur due to the faulting of a particular byte, that is the Z1W fault model in many cases induces the same fault as the Z1B by setting a following byte at a later address to zero. Thus, the lack of gaps in the vulnerabilities and the lack of a second group of incorrect results implies that while there is some coincidence, the Z1W fault model does not match the EMP effects well.

The NOP fault model is similar to the Z1B fault model in having groups of vulnerabilities that match very well with the hardware experiments. The lack of two groups of incorrect results however implies that the NOP fault model does not accurately represent the EMP effect on the hardware.

Considering combinations and subsets of the fault models is straightforward from the above results and for the Z1B and NOP fault models applied only to the first byte of each instruction those displayed in Fig. 10.3. Observe the two fault models now matche but that no single fault model alone exactly matches the hardware results. Considering the Z1B and NOP instructions combined (or combined, but taking only the NOP targeting the first byte of each instruction) provides the closest match to the hardware results.

10.2 Backdoor

This section presents the Backdoor case study experimental results. Section 10.2 presents the software experimental results for the backdoor case study. Section 10.2 highlights the hardware experimental results. Section 10.2 overviews the comparison of the software and hardware results. Note that the experimental methodology here is the one in Chapter 9.

Software-Based Experimental Results

This section overviews the results of the software simulation experiments.

An overview of the backdoor case study software experiment results can be seen in Figure 10.4. Observe that all the fault models indicated possible vulnerabilities around bytes `80005ca` and `800061a`. The FLIP fault model indicated additional vulnerabilities at byte `80005a7`.

The vulnerabilities detected at bytes `80005ca`-`800061a` by all but one fault model correspond to the `POP` instruction at `80005ca` in Listing 8.4.

Similarly to the control flow hijacking case study results, the effect observed on the assembly instructions is classified into two different effects (see Section 10.1). The

first effect is when the fault injection simulation changes the original assembly instruction to a different assembly instruction or multiple assembly instruction. The second effect is when the fault injection simulation changes the original assembly instruction address, value, or register to a different values keeping the same assembly instruction.

The rest of this section explains the effects of the fault injection simulation using the different models on the two instructions on byte addresses `80005ca` and `80005a7` in Listing 8.4.

The instruction `bd80 pop {r7, pc}` at byte address `80005ca` in Listing 8.4 stores the top value of the stack into registers `r7` and `pc`. This instruction indicates the end of the `blink_wait` function. The simulation of the fault injection using the fault models produces the following effects:

— Using all the Z1B, Z1W, and FLIP fault models, it is possible to change the `POP` instruction to `LSL`, `MOV`, `ADD`, ... instructions.

— Using the NOP fault model, it is possible to replace the instruction with a `NOP` instruction.

— Using the FLIP fault model, it is possible to modify the registers that will be modified after the pop. Here instead of loading values of the stack into registers `r7` and `pc`, it will only load the value into register `r7`.

All the above modifications will skip the execution of the `POP` instruction, and so execute the `wait_for` value corresponding to a branching instruction to the `backdoor` function.

An interesting vulnerability which was detected at byte `80005a7` by the FLIP fault model, corresponds to the instruction `b.n 80005b4` at `80005a6` in Listing 8.4. This instruction is a branching instruction, which will jump to the instruction at `80005e8` in Listing 8.4. The effect of (simulated) fault injection using the FLIP fault model was to change the target address of the branch directly to the `backdoor` function.

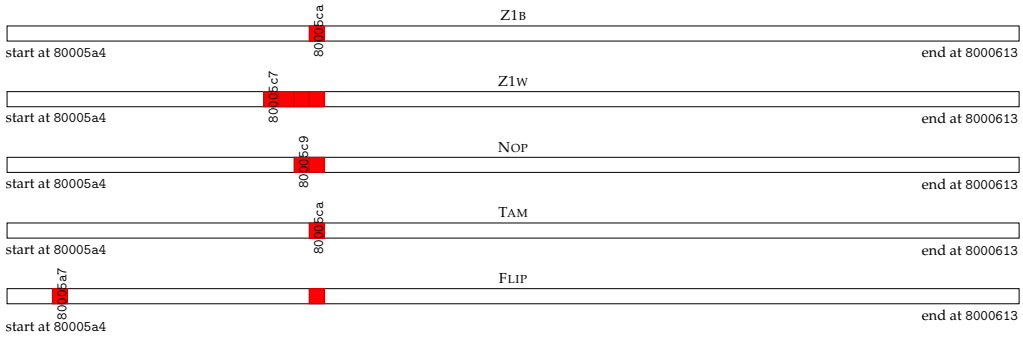

FIGURE 10.4 – Software-Based Backdoor Results

Hardware-Based Experimental Results

This section overviews the results of the hardware experiments.

An overview of the backdoor case study hardware experiment results can be seen in Figure 10.5. As before (see Section 10.1) various measurements and experiments were performed to ensure the correct timing for the fault injection, and a large number of experiments were run to yield the results. Observe that the only vulnerabilities were detected between 1.224μs and 1.260μs.

By calculating the execution time for the instructions, clock cycles, hardware latency, etc. the fault injection at time 1.224μs to 1.260μs corresponds to the `POP` instruction at `80005ca` in Listing 8.4.

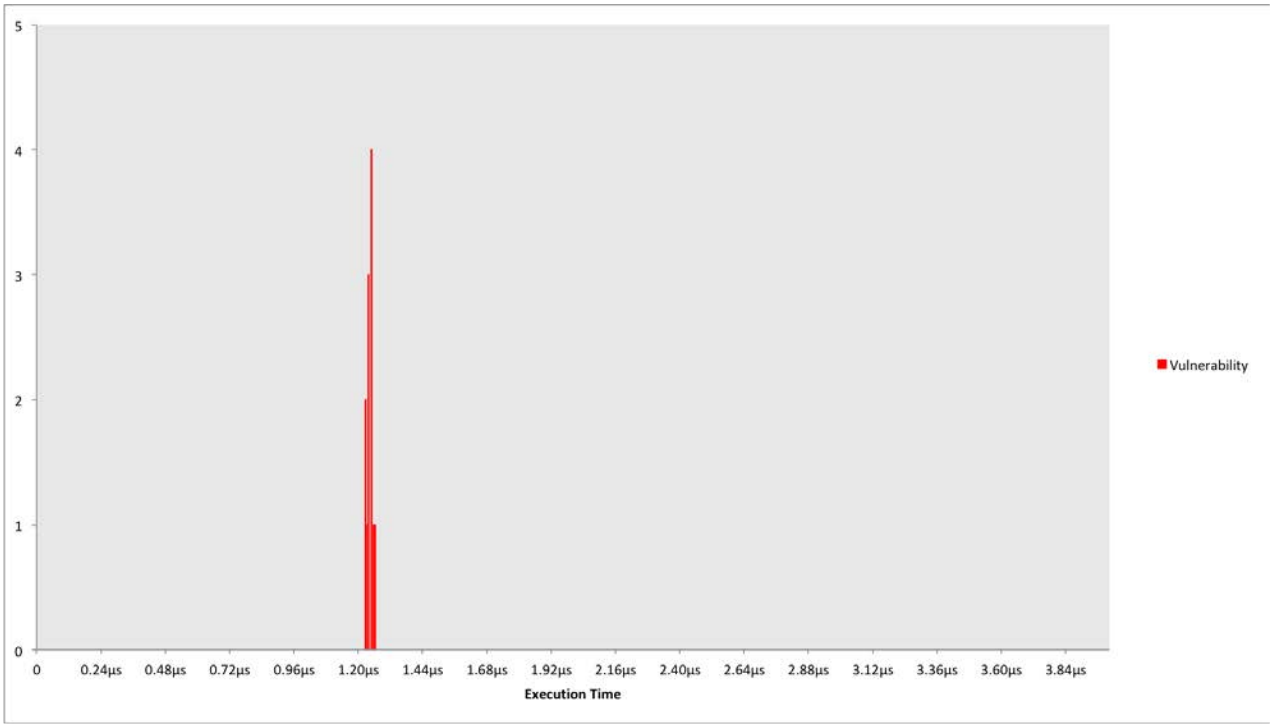

FIGURE 10.5 – Hardware-based Backdoor Results

Comparison

This section compares the results of the software-based and hardware-based fault injection experiments from the previous two sections (10.2 & 10.2). Both approaches detected vulnerabilities in the instruction at byte addresses `80005ca` in Listing 8.4.

Due to the very limited hardware results (only a single spike of vulnerabilities and no incorrect result), the comparison is both trivial and less interesting. All the fault models were able to detect a vulnerability in the instruction at byte address `80005ca` in Listing 8.4.

— The Z1B and TAM fault models detected a fault injection vulnerability at the exact same address as the hardware approach and nowhere else.

— The NOP fault model also found a fault injection vulnerability at `80005c9` since the NOP fault model changes the value of two bytes and so will impact the instruction at byte address `80005ca`.

— The Z1W fault model found faults at four byte addresses `80005c7` to `80005ca`, but in practice this was merely due to the size of the fault model, since all Z1W faults starting from `80005c7` set the byte `80005ca` to zero.

— The FLIP fault model was the only one to have a significant difference also finding a fault injection vulnerability in the instruction at byte address `80005a7` in Listing 8.4.

The comparison here offers little useful information in improving the understanding of the relation between software-based and hardware-based approaches. The ruling

	Backdoor		CFH	
Fault Model	Vulnerabilities Detected	Runtime	Vulnerabilities Detected	Total Runtime
Z1B	2	4m51s	7	2m14s
Z1W	4	4m55s	9	1m52s
NOP	3	5m18s	7	2m39s
TAM	10	4m31s	3	1m42s
FLIP	14	45m55	40	16m51s

TABLE 10.2 – Experiment Runtime

out of the FLIP fault model as being likely for EMP effects aligns with the results of Section 10.1, but little else was learned here that can improve over the information gained from Section 10.1. (That said, the agreement on the FLIP fault model and lack of contradiction at least supports the prior conclusions.)

Runtime Information

Table 10.2 shows the experiment runtime for each fault model of the two case studies backdoor and CFH. Notice that the runtime is relatively the same for each fault model except for the FLIP fault model. This difference is due to the number of verified mutants. For the backdoor case study, the number of mutants generated after the fault injection simulation using the Z1B, Z1W, NOP, and TAM was 116 mutants, compared to 928 mutants for the FLIP fault model. Similarly for the CFH case study, 44 mutants for each of the first four fault model, compared to 352 for the FLIP fault model. Overall, the verification of a single mutant requires approximatively 3 secondes.

Chapter 11

Details on the Experimental Results and Implementation for the CFH Case Study

This chapter demonstrate in details how the software-based approach results presented in Chapter 10 were obtained by following the steps of the FIVD process presented in Chapter 9.

Property Specification

This chapter takes the control flow hijacking attack as an example. Listing 11.1 presents the C source code of the target function to verify. In a normal execution not given the false pin, the function should output the value `0x00000000`. A vulnerable behaviour will be detected if the function output the value `0x55555555` given the correct pin.

```
1  uint32_t test_persistence (void){
2   HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7, GPIO_PIN_SET);
3   uint32_t status = 0;
4   if (pin_correct==1) {
5     status=0x00000000;
6   } else {
7     status=0x55555555;
8   }
9   HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7, GPIO_PIN_RESET);
10  return status;
11 }
```

LISTING 11.1 – Control Flow Hijacking Case Study C Code

At the assembly level presented in Listing 11.1, notice that the if condition (at line 4 in Listing 11.1) is represented at the compare instruction at address `8000ab4` in Listing 11.2. The condition `(pin_correct == 1)` is checked and then the `status` value is loaded in register `r4` (lines 13 and 14) based on the comparaison results.

```
1  08000aa0 <test_persistence>:
2   8000aa0: b510 push {r4, lr}
3   8000aa2: 480a ldr  r0, [pc, #40]
4   8000aa4: 2180 movs r1, #128
5   8000aa6: 2201 movs r2, #1
```

```
8000aa8: f001 f91c bl 8001ce4 <HAL_GPIO_WritePin>
8000aac: 4b08 ldr  r3, [pc, #32]
8000aae: 4807 ldr  r0, [pc, #28]
8000ab0: 681b ldr  r3, [r3, #0]
8000ab2: 2180 movs r1, #128
8000ab4: 2b01 cmp  r3, #1
8000ab6: bf0c ite  eq
8000ab8: f04f 34ff moveq.w r4, #0; 0x00000000
8000abc: f04f 3455 movne.w r4, #1431655765; 0x55555555
8000ac0: 2200 movs r2, #0
8000ac2: f001 f90f bl 8001ce4 <HAL_GPIO_WritePin>
8000ac6: 4620 mov  r0, r4
8000ac8: bd10 pop {r4, pc}
8000aca: bf00 nop
8000acc: 40011000 .word 0x40011000
8000ad0: 2000001c .word 0x2000001c
```

LISTING 11.2 – Control Flow Hijacking Case Study Assembly Code

The specified property here will check if the register `r4` can eventually have the value `0x55555555` (which is equal to `#1431655765`). The property is expressed in the BLTL form as follow:

```
F<=#1000 ( r4=1431655765)
```

LISTING 11.3 – Control Flow Hijacking Property

— `F` is the temporal operator for eventually,

— `<=#1000` expresses the bound, it gives the length of the run (number of steps) on which the property must hold, since here the property is expressed in the bounded linear temporal language (BLTL).

— `( r4=1431655765)` checks if the output value could be 0x55555555. Note that under normal execution and valid pin this case cannot happen without any fault injection.

The property is written in a separate file with the extension `.bltl`. This file will be used by the model checker tool (Plasma Lab) to verify the generated mutants. **If the property holds on the model, this means that the fault injection created a vulnerability, and that the control flow of the program is hijacked.**

Obtaining the Results

As it was explained in Chapter 9, in order to obtain the results the FIVD process implementation was adopted (see Figure 9.2, page 84).

The bash script in Listing 11.4 shows in a summarised way the FIVD process implementation (Note that priorly the validation step is done). The SIMFI is used first to simulate the fault injection with the chosen fault model and generate the mutant. Then, the mutant is translated to an RML model using the ARML tool (further information on the RML model will be given later). After, Plasma Lab model checker is used to check if the specified property (see Listing 11.3), holds on the generated model by the RML tool (see Listing 11.5)

```
#1- Fault injection Simulation using SimFI
python ../faultinject.py -f $faultmodel -a $i -o
    $BinaryMutantName -i $bianryName
#2- RML generation using ArmL
./translation_tool $BinaryMutantName $RmlMutantName
    $startAddress $binarysize
#3- Check the property using Plasma
./plasmacli.sh launch -m $RmlMutantName:rml -r $propertyfile:
    bltl -a montecarlo -A"Total samples"=100 --progress
```

LISTING 11.4 – Bash Script

RML Model

This section presents the RML model generated by the ARML tool. Listing 11.5 shows a part of the RML model corresponding to the assembly code in Listing 11.2 (headers information and variables declaration is not shown here).

Each instruction in Listing 11.2 is translated to the corresponding one in RML model. Each line in Listing 11.5 represent a transition in the model. Let's take as an example the first line in Listing 11.5:

- `pc=66562` is the *guard* in the RML language, it represents the predicate over all the variables in the model. In other world this is the current state of the model, which corresponds here to the program counter.
- `-> 1` is the *probability* of the transition, it express the probability that the model will take this transition.
- `(pc' = 66564) & (r0' = M0)` is the *update* done by the transition if the *guard* is true. Here the value of `pc` will be updated to the address of the next instruction, and the register `r0` will receive the value of the memory address `M0`.

```
1  [] pc=66562 -> 1 : (pc' = 66564) & (r0' = M0);
2  [] pc=66564 -> 1 : (pc' = 66566) & (r1' = 128);
3  [] pc=66566 -> 1 : (pc' = 66568) & (r2' = 1);
4  [] pc=66568 -> 1 : (pc' = 66572) & (lr' = 66572);
5  [] pc=66572 -> 1 : (pc' = 66574) & (r3' = M1);
6  [] pc=66574 -> 1 : (pc' = 66576) & (r0' = M1);
7  [] pc=66576 -> 1 : (pc' = 66578) & (r3' = M0);
8  [] pc=66578 -> 1 : (pc' = 66580) & (r1' = 128);
9  [] pc=66580 -> 1 : (pc' = 66582) & (flag' = ( r3 = 1 ) ) ;
10 [] pc=66582 -> 1 : (pc' = 66584) ;
11 [] pc=66584 & flag=true -> 1 : (pc' = 66588) & (r4' = 0);
12 [] pc=66584 & !flag=true -> 1 : (pc' = 66588) ;
13 [] pc=66588 & flag=false -> 1 : (pc' = 66592) & (r4' = 1431655765);
14 [] pc=66588 & !flag=false -> 1 : (pc' = 66592) ;
15 [] pc=66592 -> 1 : (pc' = 66594) & (r2' = 0);
16 [] pc=66594 -> 1 : (pc' = 66598) & (lr' = 66598);
17 [] pc=66598 -> 1 : (pc' = 66600) & (r0' = r4);
18 [] pc=66600 -> 1 : (pc' = 66604) & (sp' = 8);
19 [] pc=66602 -> 1 : (pc' = 66604) ;
```

LISTING 11.5 – Control Flow Hijacking RML Model

Understanding the results

Listing 11.6 highlights in red colour all the instructions which, when modified during the fault injection simulation, created vulnerabilities.

To understand why the fault injection simulation created a vulnerability, a closer look to one of the instruction is given. For example, the instruction in line 11 Listing 11.6 compares the value of the register `r3` to 1 and sets the flags based on the obtained result.

Next, the effect of the different fault model on the instruction `cmp r3, #1` in line 11 Listing 11.6 is given:

Z1B zeroing the byte address `8000ab5` will produce the following instruction `cmp r3, #0`.

NOP nopping the instruction at address `8000ab4` will replace the compare instruction with a nop instruction `nop`.

TAM replacing the byte address `8000ab5` will produce the following instruction `cmp r3, #ff`.

FLIP flipping the fifth bit in the byte at address `8000ab4` will produce the following instruction `cmp r7, #0`.

All the changes presented above will modify the control flow of the program. The modification in the compare instruction will have an impact on the chosen path afterwards Note that the list above did not show all detected vulnerabilities.

```
1  08000aa0 <test_persistence>:
2   8000aa0: b510 push {r4, lr}
3   8000aa2: 480a ldr  r0, [pc, #40]
4   8000aa4: 2180 movs r1, #128
5   8000aa6: 2201 movs r2, #1
6   8000aa8: f001 f91c bl 8001ce4 <HAL_GPIO_WritePin>
7   8000aac: 4b08 ldr  r3, [pc, #32]
8   8000aae: 4807 ldr  r0, [pc, #28]
9   8000ab0: 681b ldr  r3, [r3, #0]
10  8000ab2: 2180 movs r1, #128
11  8000ab4: 2b01 cmp  r3, #1
12  8000ab6: bf0c ite  eq
13  8000ab8: f04f 34ff moveq.w r4, #0; 0x00000000
14  8000abc: f04f 3455 movne.w r4, #1431655765; 0x55555555
15  8000ac0: 2200 movs r2, #0
16  8000ac2: f001 f90f bl 8001ce4 <HAL_GPIO_WritePin>
17  8000ac6: 4620 mov  r0, r4
18  8000ac8: bd10 pop {r4, pc}
19  8000aca: bf00 nop
20  8000acc: 40011000 .word 0x40011000
21  8000ad0: 2000001c .word 0x2000001c
```

LISTING 11.6 – Control Flow Hijacking Case Study Assembly Code - Highliting the Results

Chapter 12

Discussion and Limitations

This chapter concludes the second part of this thesis. The first section of this chapter discusses the experimental results obtained by software-based and hardware-base approach, and how combining the two approaches will have benefits compared to applying only the software or the hardware approaches. The second section of this chapter investigates the limitation of combining the software-based and the hardware-based approches.

Sommaire

12.1 Discussion

Case Studies Results

This section discusses the experimental results presented in chapter 10 and what we can learn from them.

By comparing the software and hardware experimental results it is possible to determine which software fault models best correspond to the EMP effects observed. Here the Z1B, Z1W and NOP fault models had the closest correlation with the observations of the EMP faults induced. To some extent this agrees with previous work [96] that observed that the most accurate fault model is an instruction skip (or here NOP). However, there is also strong evidence from this work that other fault models, in particular setting all of a byte or word to zero (i.e. Z1B or Z1W), also correlate strongly with the effects of EMP.

Observe also that **the software vulnerabilities generated using the** FLIP **and** TAM **did not correspond well to the EMP fault injection results**. Although the FLIP and TAM detected some similar vulnerabilities to the other fault models, both fault models also detected a lot of vulnerabilities which do not correspond to the hardware results. In particular the TAM fault model never produced an incorrect result despite many being observed in other fault model, and the FLIP fault model had many vulnerable or incorrect results that did not correlate with the EMP results.

Using the software results to learn about the hardware results is also possible. The hardware experiment results do not indicate *how* the fault was achieved or what the actual fault model/effect was, only the outcome. Knowing the specific effect of the fault injection on the hardware is a nontrivial task, specially when using imprecise

hardware techniques such as EMP. Hardware experiment results alone are only able to show that the injection of the fault creates the desired vulnerability, but do not give detailed information of what, where, or how the injected fault created the vulnerability. The results here indicate that the strongest correlation is with instructions simply being faulted to have alternate or no effect (i.e. the Z1B, Z1W, and NOP fault models). Further, since none of these fault models correlates exactly, this implies (along with the inconsistent nature of achieving a vulnerable or incorrect outcome) that EMP fault effects may vary and not have a single fault model.

From the experiment results one can observe that **the hardware and software results do coincide but they do not exactly match**. There are clearly locations in the assembly code where many fault models indicate a vulnerability (or incorrect result) and these correlate very strongly with the locations where the hardware experiments were able to produce vulnerabilities (or incorrect results, respectively). This clearly indicates that there is a coincidence between the software-based and hardware-based approaches.

Considering the results further, one key insight is that the **software-based experiments did not have any false negatives**. That is, every place where the hardware was able to produce a genuine vulnerability (or incorrect result), the software-based approaches indicated a vulnerability (or incorrect result, respectively) for at least one fault model. (Indeed, this holds even when only considering the Z1B, Z1W, and NOP fault models.) Thus, absence of any vulnerabilities or incorrect results according to software-based experiments implies that no such vulnerabilities or incorrect results should exist in practice.

The software-based approach does produce false positive results. This outcome is not surprising since many fault models were tested here, including ones unlikely to be possible with the hardware-based EMP fault injection. However, even when considering only the Z1B, Z1W, and NOP it is not clear that *every* vulnerability or incorrect result can be reproduced by the EMP experiments. **The conclusion here is that software-based simulations can find vulnerabilities (or other behaviours) that may be infeasible to reproduce in the hardware, or at least extremely difficult to achieve**.

From all the above, one can conclude that: on one hand software alone is not sufficient to claim that vulnerability exists and is real, on the other hand hardware alone is not feasible to explore all the possible configurations and locations in the target program. That is, the software can be quickly used to find many potential vulnerabilities (or other results) even on relatively large programs (up to 800 line of code), but that these cannot be guaranteed to exist in practice. The hardware can guarantee a vulnerability (or other outcome) when one is produced, but finding these is extremely expensive in time and equipment.

Combined approach

The natural extension of these hardware and software results is to consider how they could be combined. This section discusses how this can be achieved to rapidly find genuine vulnerabilities that would be infeasible with either approach alone. Observe that this approach does *not* rely upon any prior knowledge of weaknesses in the code.

If only the software-based approach is used then although the results are quick to compute and require only a moderate amount of computational resources (this is

relative to the model checker performance regarding the program size), there is not guarantee that any of the results hold. Indeed, attempting to address too many false positives would be intensive on developer resources and a waste of effort if the vulnerabilities are not genuine.

If only the hardware-based approach is used this is extremely expensive if not infeasible to test larger programs. This requires many experiments to test each possible timing/location of fault injection on the program over the programs entire execution life-cycle, which may be impossible for programs designed to run for years.

The proposed combined approach is to use the software-based simulations to quickly find all the *potential* vulnerabilities in a given program. This can be easily applied and automated [54, 55] to yield information on all the locations in the code that may be vulnerable. **The hardware-based approach can then be applied to test the most vulnerable locations to rapidly confirm (or refute up to some margin of confidence) the existence of the vulnerability**. In practice this requires some small amount of computational resources for the simulations, and then only limited time and some calculation prior to testing with the hardware to accurately target the right locations.

The rest of this section explores how the above combined approach could have been applied to the case studies here, and demonstrates the efficacy of the combined approach.

For the control flow hijacking case study, $\sim$ 117035 hardware experiments were conducted to generate the results shown in Fig. 10.2. (This number accounts only for experiments after calibration, latency tests, etc.) Overall, these experiments indicated a vulnerability 0.469% of the time, and only in certain locations. Thus, to find one requires some significant investment in time to scan the entire function and test each location frequently enough to be likely to find a genuine vulnerability. However, if the software experiments are used to guide the hardware experiments, it is possible to target exactly the timing 1.344μs, which could then demonstrate a vulnerability with 0.999 probability requiring only 10 passes over 21 timings (total 210 experiments). **Thus, this approach could bring the number of hardware experiments required down orders of magnitude and still confidently confirm or refute a fault injection vulnerability**.

For the backdoor case study the possibility to find the vulnerability using hardware alone is significantly lower since the location is unique and has a low probability of success. Overall the combined probability of both targeting the right timing and inducing a fault for any given experiment is 0.00957%. However, if guided by the software results that all indicated a specific location to test (i.e. 1.248μs) then the probability to detect a fault is 0.999.

Observe that in both case studies (Backdoor and CFH) vulnerabilities were already expected and the locations could be guessed or calculated in advance. However, using the combined approach described here does *not* require this prior knowledge since the software simulations can be performed to find the likely locations to confirm or refute with hardware experiments.

This means that **there is no need to know in advance whether a fault injection vulnerability exists. The software can be used to locate any potential fault injection vulnerabilities, and the hardware used to confirm of refute their feasibility of exploitation**. This combined approach is more accurate than software simulations alone (since the false positives are refuted), and much cheaper than the hardware

alone since many less experiments are required to demonstrate or refute vulnerabilities.

12.2 Limitations

This section discuss the limitation of combining the software-based and hardware-based fault injection approaches. The limitations of this second part are resumed in three limitations.

The first limitation is mapping between the software-based and hardware-based fault injection approaches. In [39] the authors concluded that the software and the hardware results do not map because of the fault detection time. This is because in hardware-based approach there is a delay between the injection of the physical fault and the observed effect, in the software-based approach the effect of the injected fault is observed instantly. To overcome this problem in this thesis a computation of this delay was done (as shown in Section9.3). The computed delay was taking into consideration while mapping the software and hardware result, combined with the number of clock cycle per instruction and the clock cycle timing.

But the two case studies (Backdoor and control Flow Hijacking) presented in this thesis are considered to be a proof of concept and do not correspond to real program implementation. The mapping task for the two case studies was feasible but will not be scalable to a larger program.

The second limitation is related the implementation of the software and hardware approach to a limited architecture. Both existing software-based and hardware-based implementation are not generalised to all possible architectures.

The presented software-based approach implementation of the FIVD process in this thesis is related to a chosen architecture. The FIVD process works on the binary which is related to specific architecture instruction set. In this thesis is was shown that the process can be implemented in two different architecture (x86 and ARM). Having a general implementation of the process that supports all the architecture set is not feasible. The FIVD is a general approach to different architectures but no general implementation exist.

The hardware-based approach can be applied to devices with different architectures. The parameter set for hardware experiment on a specific chip architecture can not be generalised to other chip architecture, therefore the implementation of the hardware-based approach can not be generalised to all architectures. The hardware-based approach used in this thesis was limited to a single technique (EMP) and a single architecture (ARM), to give a proof of concept. Parameter experiments were set to a chosen target and can not be used for a different one. But the hardware process can be considered as a general one to other kind of architecture.

The third limitation is about the specification of properties. In the software-based approach proposed in this thesis, the properties are not generated automatically and need some expertise to be specified. If the properties are not well defined the vulnerability detection can not be done correctly. Property specification is related to the program (system) design/functionalities/components/variable, therefore **it is not feasible to have general property valid for all programs (systems)**. The detection of the vulnerability is related to the property, if the property is not specified properly the vulnerability can not be detected. It is very important in the proposed approach in this thesis to specify the properties correctly.

Conclusions and Future Work

Chapter 13

Conclusions

Fault injection represents a serious threat to the robustness and security of many software systems used in daily life. There are two main approaches to detect fault injection vulnerabilities and to test system robustness: software and hardware-based approach. Both approaches yielded useful results and can make useful contributions. Software-based approaches are good for simulation and being able to cheaply implement, albeit at the cost of the ability to demonstrate a fault injection vulnerability is genuine and can be exploited. Hardware-based approaches are good for proving genuine exploitability, but are expensive in time, equipment, and expertise to conduct.

In Part I of this thesis the focus was on the software-based approach. To advance the current state of the art of the software-based approaches, this thesis proposed the FIVD process. FIVD is an automated formal process that uses model checking to detect fault injection vulnerabilities in binaries. The FIVD process is generic, in the sense that it is automated, operates at the binary level, supports the detection of many varieties of fault injection vulnerabilities, and does not rely on any particular system architecture, fault model, or other restricted choices (as are common in the literature).

Through this thesis, the FIVD process was extended to detect fault injection vulnerabilities in larger programs by adding the pre-analysis step. The addition of pre-analysis to the process allowed many fault injections to be easily ignored as yielding failed program states, thus improving the efficacy of the process as a whole. Pre-analysis step reduced the number of mutants requiring model checking by 66.33%. Overall the process is scalable via parallelism, although model checking remains expensive.

The FIVD was first applied to the motivating example as a proof of concept. The experimental results showed several vulnerabilities in the verifyPIN program. From the experimental results it was shown that:

— it is important to specify the property that will catch any change of the normal behaviour of the program, since if not specifying a generic property not all the vulnerabilities will be detected.

— the property had an impact on the time needed to perform model checking, which impacts the time needed to get the results.

To show the scalability and efficiency of the extended version of the FIVD process, it was applied to the cryptographic algorithms (PRESENT and SPECK). The application of the FIVD process on the cryptographic algorithms case studies detected 82 vulnerabilities, 73 in PRESENT and 9 in SPECK. Both PRESENT (9 vulnerabilities) and SPECK (4 vulnerabilities) were vulnerable to faulting jump instructions

that allowed encryption to be entirely bypassed. For SPECK five vulnerabilities exist, two by inserting non-operation byte values, and three by flipping individual bits. PRESENT was also found to be vulnerable to 64 different CA vulnerabilities, mostly through bit flips, but also through nopping instructions and jump target modifications. These were all found towards the end of the binary, where the last round of encryption occurs. SPECK was not found to be vulnerable to the CA vulnerabilities tested here. This is mostly due to the number of rounds (21) to achieve encryption and that 21 is not one bit flip away from either 9 or 10. These results indicate that:

— some kinds of attacks may be more or less achievable depending on the structure of the code used, and so some care should be taken when choosing how to implement a fault resistant binary.

— programs tend to be more vulnerable to bit flip and modifying the jump address faut model.

Despite the good results of the software approach, it is hard to guarantee that the detected vulnerabilities correspond to real vulnerabilities in practice. The fact that the fault injection are simulated gives no guarantee that a physical fault injection attack on the system will have the same effect. Thus the challenge in Part II of this thesis to combine both software and hardware based approaches on a single case study to explore how the two approaches connect.

Both software-based and hardware-based approaches have been used to detect fault injection vulnerabilities. However, the two approaches have not been directly compared before. Part II of this thesis presents both broad spectrum software-based formal methods analysis and large scale hardware based experiments performed on the same case study. The results of these experiments are compared to explore what can be learned by bridging the two approaches.

For the software-based approach an improved version of the FIVD process is used. Contrary to the version presented in Part I the properties are no longer added to the source code, now the properties are defined in a separate file called during the validation and checking steps. The implementation of the FIVD process is no longer limited to the X86 architecture and bounded model checking.

The results of Part II show that software-based approaches do find genuine fault injection vulnerabilities. Although software-based approaches may suffer from some false positives, they (when done with multiple fault models) *do not have any false negative results*. This allows for software-based approaches to provide useful information about potential fault injection vulnerabilities, and strong guarantees about the absence of fault injection vulnerabilities.

The results of Part II also showed that (confirming prior work [95]) EMP effects do not have a single fault model. The results here indicated that multiple fault models together best represent the effects of EMP fault injection attack. In practice, these fault models correspond to an EMP effect either wiping a byte or word (by setting all the bits to zero) or skipping an instruction.

More generally the results show that there is a strong relation between the software and hardware-based approaches. This gives support to research that uses software-based approaches to simulate or approximate hardware experiments. Further as mentioned above, the coincidence can be used to influence our knowledge about both approaches and refine our understanding of them.

Combining both software-based and hardware-based approaches is also vastly more effective in isolating and confirming the existence of a fault injection vulnerability.

In practice, by combining both approaches, finding previously unknown vulnerabilities on whole programs becomes feasible. In the future this should allow the much more rapid discovery of genuine fault injection vulnerabilities that do not require prior knowledge or intuition on the part of the researcher.

Chapter 14

Future Work

Although this thesis advances the state of the art in software-based fault injection approach and combined software and hardware-based approaches in various ways, there are still many opportunities for progress and areas that need significant effort to be able to make fault injection vulnerability detection a reliable and easily applicable part of software and hardware development. This chapter discusses possibilities to expand on the work presented in this thesis.

Formal Verification part

Another area to advance in would be, the application of formal methods. This thesis exploited the use of bounded and statistical model checking to overcome one of the major problem of model checking: the space state explosion. Still the use of model checking here is not optimal and requires a lot of time. The fact that all the generated mutants are model checked is not such a good idea. Thus, an incremental approach may yield significant efficiency returns. Similarly, developing and exploiting formal methods that focus on the exact problems considered in vulnerability detection could yield much more precise results than those that are currently state of the art.

Regarding properties, another area of future work is to consider how to extract properties automatically from the binary (or source code). There is some existing work in this area [122, 153] although they focus on high level behaviour rather than binary code.

Currently the process identifies vulnerabilities, but does not suggest fixes or countermeasures. Automatically generating countermeasures is non-trivial, although if countermeasures to particular faults are known future work could suggest or implement them automatically. Perhaps more significantly, these countermeasures could be checked immediately using the process here and so their effectiveness verified immediately.

Bridging Software and Hardware based approaches

Work on strongly connecting the software and hardware-based approaches is clearly a goal for future research and development. As it was mentioned before only few research works until today were interested in combining the two approaches, thus this research area is to be more exploited.

A first direction is to focus on improving the software-based process as explained in the previous points. The software-based approach is the main interest in this thesis.

In order to go further there is a need to have a more robust and reliable software process that will be combined with the hardware one.

The case studies used in the experiment for this thesis showed that combining the software and hardware approach is promising. An other direction would be to run software and hardware experiments on other security critical software, e.g. encryption algorithms, mission critical software, embedded device kernels, and also software that has implemented countermeasures.

Related also to the experiment, in this thesis the software results were compared only to the EMP hardware technique. The chosen hardware technique was limited to the existing equipments in the LHS lab. An interesting direction would be to run further experiments using other techniques (laser, power supply, etc), in order to compare the results of the different hardware techniques with the results of the software one.

A strong foundation of understanding of the relations between different kinds of software and hardware based approaches will enrich and improve the results of both. Further, by connecting these results, software-based results can be validated to be genuine by reproducing them with hardware experiments. In the other direction, hardware-based experiments will demonstrate the efficacy and accuracy of the software-based approaches.

Finally, many existing works in the domain of fault injection vulnerabilities and their detection work on examples or programs where a vulnerability is already known to exist. The goal of the work is to (re)produce a known attack (or exploit one that has been intentionally designed in) to demonstrate the efficacy of the approaches used. However, finding vulnerabilities that were not even suspected in advance, or devising approaches that allow the finding of such vulnerabilities in an efficient manner is a clear requirement for practical application in the future.

Engineering of the developed tool

The FIVD process proposed in this thesis required a lot of engineering efforts. First, the efforts of putting together in a tool chain existing tools that had limitations. Then, the efforts in developing new tools to satisfy the requirement of this thesis.

SIMFI is a tool that simulates fault injection on binaries at the chosen location and with the chosen fault model. At the state of submitting this thesis, the SIMFI tool is considered to be sufficient to give reliable results, but there remains much room for further improvement. Several tools to simulate fault injection attacks on software exist [67, 71]. However, these tools are limited by various choices that make unsuitable for the process here (hence their lack of use in the implementation). Several are only able to inject faults into intermediate representations, and not into executable binaries [110, 135, 142], thus being unable to simulate faults that appear only at the executable binary level. Others have different limitations, such as: specific hardware platforms [107, 120], specific source code languages [35, 89, 142], or requiring simulating drivers [37]. Despite these limitations, many include useful techniques or developments that could be incorporated into future development of the SIMFI tool.

ARML is a tool that generate the RML model from an ARM binary file. In Part I if this thesis, FIVD process was implemented using MC-Sema [136] that has various limitations with instruction sets, or failures to correctly translating the behaviour of the program. Similarly, in many other works [65, 107] the tools limit the applicability

of the technique to some limited scope, limited architecture, limited size, etc. Thus, in many areas the tools used require refinement and maturity, and in other areas the tools simply do not exist and would need to be created, thus the development of the ARML tool. At the state of submitting this thesis the ARML tool is considered to be a first version but is not matured yet. It does not support all instructions set of the ARM architecture yet. But this is straight forward to add and will not require any change to the architecture or conceptualisation of the tool.

Appendix A

FIVD Process Implementation Details

This section details the FIVD process implementation presented in Section 4.2.

Source Code & Properties

The implementation starts with the source code written in the C language, including the properties to be validated and checked that are expressed as assert statements in this source code. For example, the source code of the motivating example (Figure A.1) could have the following property.

```
__llbmc_assert(i == 4);
```

that the loop counter `i` reaches `4` inserted between lines 9 and 10 in Figure A.1. This would check that `i` reaches the value `4` before doing the conditional to test whether access should be granted on lines $10 - 12$.

```
bool grantAccess = false;                                    1
bool badValue = false;                                       2
int i = 0;                                                   3
while (i < PINSize) {                                        4
  if (PINCandidate[i] != PINTrue[i]) {                       5
    badValue = true;                                         6
  }                                                          7
  i++;                                                       8
}                                                            9
if (badValue == false) {                                     10
  grantAccess = true;                                        11
}                                                            12
```

FIGURE A.1 – Motivating Example Code

Compilation

The compilation from source code to executable binary for this paper is done with GCC. Note that here a listings file is also generated with annotations that will be exploited to do the fault injection later. The following is the command used to compile with GCC.

```
$gcc -m32 -ggdb -c -Wa,-a,-ad -o test.o
     test.c > test.lst
```

Here `-m32` specifies compiling for 32-bit architecture. The `-ggdb` argument includes debugging information that will be used to help translate the intermediate language later in the implementation. The `-c` argument indicates to compile and assemble the source code, but do not link (this simplifies the scope of checking since no library code is linked at this stage). The `-Wa,-a,-ad` argument specifies annotations to output that will be used later to do the fault injection (`-a` to turn the listing on, `-ad` to omit unnecessary debug information). The `-o` is used to specify the output file (`text.o`) for the executable binary. Here `test.c` is the source file with properties. Lastly, `> test.lst` outputs the annotations used later to do the fault injection into the file `test.lst`.

Note that the above command preserves the assert statements along with the debugging information, so these can be exploited in later stages.

Intermediate Representation

The translation from executable binary to LLVM-IR is done by MC-Sema in two stages. The first stage uses the executable binary to generate a CFG. The second stage uses the CFG to generate the LLVM-IR.

Executable Binary to CFG

The first stage is done by the `bin_descend` tool (included within MC-Sema) using the below command.

```
$bin_descend -march=x86 -d
    -func-map="test_map.txt"
    -entry-symbol=checkPIN -i=test.o
```

Here the `-march=x86` argument specifies X86 architecture. The `-d` flag enables output of debugging information, used in later stages of the implementation. The `-func-map=`
`"test_map.txt"` argument informs of the file (`test_map.txt`) that contains specifications of externally referenced functions (e.g. `__llbmc_assert 1 C N` to indicate that the function `__llbmc_assert` has 1 argument, `C` to represent the calling convention here is for CleanUp to clean up the stack after the function call, the `N` to mention that the function has a return). The `-entry-symbol=checkPIN` argument indicates the function name of the entry point into the code, here the `checkPIN` function. Lastly, `-i=test.o` indicates to input from the file `test.o`.

CFG to LLVM-IR

The second stage is to translate the CFG to LLVM-IR. This is done by the `cfg_to_llvm` tool also included in MC-Sema. The command to achieve this is shown below.

```
$cfg_to_bc -mtriple=i686-pc-linux-gnu
    -driver=test_entry,checkPIN,0,return,C
    -o test.bc -i test.cfg
```

Again `-mtriple=i686-pc-linux-gnu` indicates the X86 architecture (on linux). The argument `-driver=test_entry, checkPIN,0,return,C` defines the entry point in the generated LLVM-IR to be `test_entry` and this should correspond to the entry point symbol `checkPIN` in the CFG, the `0` represents the argument count, the `return` to specify that the function has a return, and finally the `C` represents the calling convention. As usual, `-o` indicates the output file name (here `test.bc`). Lastly, `-i` is the input file name (here `test.cfg`).

Model Checking

The model checking of properties on the intermediate representation is done by LLBMC. The command used here to model check with LLBMC is as below.

```
$llbmc -function-name=test_entry
    --ignore-missing-function-bodies
     --max-loop-iterations=20
     -only-custom-assertions test.bc
```

The argument `-function-name=test_entry` makes certain that LLBMC checks the specified function (and not others). The `-ignore-missing-function-bodies` argument is used to ignore missing functions, such as those that were not linked and so do not appear in the LLVM-IR. The argument `-max-loop-iterations=20` specifies bounds for the model checking, here limited to 20 loop iterations.

The `-only-custom-assertions` argument forces LLBMC to only check the properties specified, and not other default properties. Lastly, `test.bc` is the input file.

Note that the above steps from executable binary to model checking can be repeated (with changed file names) for the mutant binary, and so will not be repeated below.

Fault Injection

The fault injection simulation is performed by editing the executable binary file, it takes an executable binary and yields a mutant binary. To achieve this the SimFI tool is used.

The SimFI tool takes as an input the executable binary file, the user needs to specify the fault model and the target bit or byte in the binary. The command used here to simulate the fault injection of zeroing one byte with SimFI is as below.

```
SimFI --faultmodel zerobyte --address 96
         --outfile test_mutant.o --infile test.o
```

The argument `-faultmodel zerobyte` specifies the fault model used to simulate the fault injection. Here the used fault model is `zerobyte` that sets the specified address byte to zero. The SimFI tool supports a variety of fault models, refer to section 4.3 to discover the other fault models. The argument `-address 96` specifies the address of the byte in the binary file to modify. Note that for the experiment a script is written to loop through all addresses in the binary file, for illustrative reason 96 ($0x60$) is chosen as target address here. The `-outfile test_mutant.o` argument is used to specify the output file, here the mutant binary. Lastly, the `-infile test.o` argument is used to specify the input file.

```
Result:
=======
No error detected.
```

FIGURE A.2 – Property 1 : executable binary verification result

```
Result:
=======
Error detected.

Error synopsis:
===============
Assertion failed: Custom assertion (assert or
   __llbmc_assert) does not hold.

Error location:
===============
Error occurs in basic block "block_0x10b" of function "
   sub_0".
No debug information available.

Stack trace:
============
#0 void @sub_0(%struct.regs* %0)
#1 i32 @demo_entry()
```

FIGURE A.3 – Property 1 : mutant binary verification result

Detecting Vulnerability

Once the results of model checking have been produced for both the executable binary and the mutant binary, fault injection vulnerabilities can be detected when these results differ. In Figure A.2 the output of LLBMC is shown for when all the properties hold. By contrast, Figure A.3 shows the LLBMC output when a property is violated. Note that due to compilation to binary and then translation to LLVM-IR, LLBMC is unable to gather sufficient information to produce a useful trace of the property violation.

Appendix B

PRESENT Experimental Results

This section presents a case study of five different fault injection attacks against the PRESENT algorithm by applying the FIVD process without the pre-analyse step.

Section 5.2 shows the result of the PRESENT algorithm using the improved version of the FIVD. Here are the first results obtained without the pre-analyse step that triggered thoughts of improving the process for larger programs.

Experimental Design

All the experiments tested a single property to capture the capability of a fault injection attack to bypass the encryption algorithm. The property checked whether the "ciphertext" at the end of the encryption was different to the "plaintext". Thus, violations of this property indicated the encryption algorithm had been effectively bypassed. The result of each fault injected mutant binary were thus classified into one of: *passed* where model checking of all properties succeeded; *infinite loop* when the fault caused an infinite loop in model checking; *crashed* when the fault caused the program to crash; and *vulnerable* when the fault caused the property to be violated.

The five fault models are: *modifying an unconditional jump* (JMP) to jump to a new address; *modifying a conditional jump* (JBE) to jump to a new address; *zero 1 byte* (Z1B) that sets a single byte to zero; *zero 2 bytes* (Z2B) that sets two consecutive bytes to zero; and *NOP'ing an instruction* (NOP) that sets a byte to a non-operation code. Each is detailed below when considering the results for that fault model.

Results Overview

An overview of the results for injecting these fault models in all possible locations in the PRESENT binary can be seen in Table B.1. All the fault models tested caused crashes, with these being most common with the Z2B and NOP fault models. Infinite loops were also quite common, either through modification of jumps, damaging iterator code, or damaging conditionals. Vulnerabilities were quite rare, which was as expected, with all arising from the jump fault models. The rest of this section considers each of the fault models and the associated experimental results in detail.

Unconditional Jump

The fault model for this experiment was to identify unconditional jump instructions and change their target. For simplicity only increasing the value of the target address

	Fault Model					Colour
Result	**JMP**	**JBE**	**Z1B**	**Z2B**	**NOP**	
Passed	1632	502	905	855	784	
Infinite Loop	106	0	53	49	93	
Crashed	62	60	172	225	248	
Vulnerable	1	8	0	0	0	

TABLE B.1 – Overview of Fault Injection Results.

was considered (i.e. jumping relatively forward, not relatively backwards). Column **JMP** of Table B.1 presents aggregate results. There are 10 unconditional jumps in the PRESENT binary at addresses 0x0120, 0x014B, 0x0155, 0x018C, 0x01D3, 0x0207, 0x02D4, 0x0313, 0x0361, and 0x0447. The only jump that yielded a vulnerability was at 0x014B, and the details are shown in Figure B.1 showing the offsets that could be jumped to and the result of checking each mutant (and the blue box indicating the end of the experiment range).

Most of the significant changes here were infinite loops, with a significant number of crashes, and a single vulnerability that skipped the entire encryption algorithm. The infinite loops are largely as expected, since the modified jump can easily skip loop iterator increment code. The crashes are also to be expected, mostly related to jumping to incorrect byte offsets for the instructions, and so yielding invalid instructions (or instructions that crash in other ways such as trying to read invalid memory segments). The single vulnerability was when the jump for the first loop of the encryption algorithm skips over the entire encryption, going straight to the end of the code. Only a single instance was found as most jumps were "short" (single byte offset), meaning they could not bypass significant amounts of code.

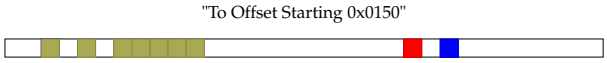

FIGURE B.1 – Unconditional Jumps from Jump at 0x014B

Conditional Jump

The conditional jump fault model changes targets similar to the unconditional jump fault model. Column **JBE** of Table B.1 presents the summaries for the two conditional jumps at addresses 0x02C9 and 0x043C. Again vulnerabilities were only found in one at 0x043C and these are detailed in Figure B.2.

Here no infinite loops were detected likely due to the conditions always being triggered at least once, instead only crashes where the unconditional jumps were instead targeting bad locations in the code leading to incorrect "instructions". More interesting are the vulnerabilities that fall into two groups. The first group (the first three in the map) jumped to later assignment instructions (including incorrectly offset locations) that ended up bypassing the correct loop controls (by changing values used

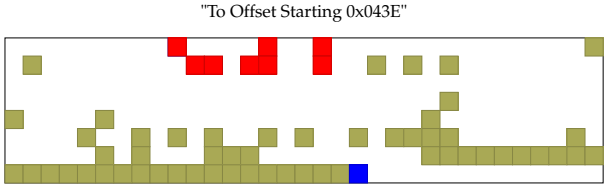

FIGURE B.2 – Conditional Jumps from Jump at 0x043C

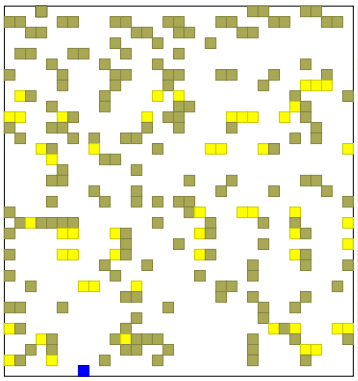

FIGURE B.3 – Zero 1 Byte

FIGURE B.4 – Zero 2 Bytes

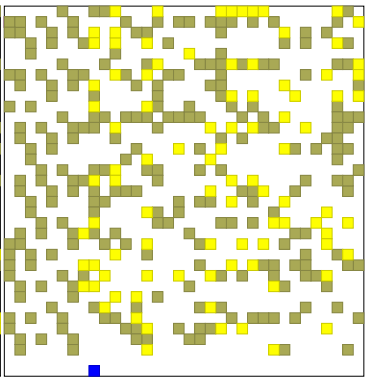

FIGURE B.5 – NOP 1 Byte

for later loop control flow), and eventually skipping the encryption algorithm. The second group (the remaining five) simply jumped to the end of the encryption code, merely bypassing the encryption algorithm.

Zero 1 Byte

Another fault model to test automating the process over a larger number of mutants was to set a single byte to zero. There are 1130 bytes in the PRESENT executable binary, and each was set to zero in a different mutant, yielding the results shown in the **Z1B** column of Table B.1. Detailed results showing which faulted bytes yield which effect can be seen in the map in Figure B.3.

No fault injection vulnerabilities to this fault model were detected, although many crashes and infinite loops were introduced. This is not a surprising result, since the PRESENT source code has two top-level loops that both perform some part of the encryption. Thus, although setting one byte to zero could skip either one of these, it would require two (non-consecutive) zero one byte fault injection attacks to be "vulnerable" here.

Zero 2 Bytes

A similar test of automation over many mutants was the fault model that sets two consecutive bytes to zero. There are 1129 possible mutant binaries under this fault model. Their results shown in the **Z2B** column of Table B.1, and the map showing the starting index of the two bytes is shown in Figure B.4.

Similar to the zero 1 byte fault injection model, no vulnerabilities were detected. Even more crashes were introduced, although a few less infinite loops. Generally this is due to instructions being damaged to yield failure, either by being simply incomprehensible, or by pushing memory access outside acceptable bounds.

NOP Code 1 Byte

The last fault model for this experiment was to set each byte to the instruction code for a non-operation (NOP). This fault injection attack was applied to each of the 1130 bytes to ensure complete coverage (and so in some cases had effects other than

NOP'ing an instruction). Column **NOP** of Table B.1 summarises these results, with the detailed map in Figure B.5.

This approach turned out to be even more destructive than either of the zero byte fault models. Although more crashes were introduced, the almost doubling of infinite loops was an unexpected result that could be investigated further in future. No vulnerabilities were detected here which aligns with the prior results that binaries are fairly resistant to these kinds of byte attacks.

A Note on Scalability

The experiments were conducted on a variety of devices with different hardware and configurations (all were virtual machines running Ubuntu X64). The distribution was due to different experiments being run at different times, however this makes it impossible to provide consistent runtime information for the experiments.

That said, in general the model checking (either validation or checking) was by far the most expensive in terms of runtime. No attempt was made to optimise or modify the settings of LLBMC to improve runtime, despite some results taking many minutes. This is due to the process being trivially parallisable, since each mutant can be checked independently.

Bibliography

[1] Farzaneh Abed et al. “Differential cryptanalysis of round-reduced Simon and Speck”. In: *International Workshop on Fast Software Encryption*. Springer. 2014, pp. 525–545.

[2] Miron Abramovici, Melvin A Breuer, and Arthur D Friedman. *Digital systems testing and testable design*. Vol. 2. Computer science press New York, 1990.

[3] Astrit Ademaj et al. “Fault tolerance evaluation using two software based fault injection methods”. In: *On-Line Testing Workshop, 2002. Proceedings of the Eighth IEEE International*. IEEE. 2002, pp. 21–25.

[4] Rajeev Alur and Thomas A Henzinger. “Reactive modules”. In: *Formal methods in system design* 15.1 (1999), pp. 7–48.

[5] Jean Arlat et al. “Fault injection and dependability evaluation of fault-tolerant systems”. In: *IEEE Transactions on Computers* 42.8 (1993), pp. 913–923.

[6] Mohamed Faouzi Atig et al., eds. *Verification and Evaluation of Computer and Communication Systems - 12th International Conference, VECoS 2018, Grenoble, France, September 26-28, 2018, Proceedings*. Vol. 11181. Lecture Notes in Computer Science. Springer, 2018. ISBN: 978-3-030-00358-6. DOI: 10.1007/978-3-030-00359-3. URL: https://doi.org/10.1007/978-3-030-00359-3.

[7] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, et al. *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science, 2001.

[8] Nasour Bagheri, Reza Ebrahimpour, and Navid Ghaedi. “New differential fault analysis on PRESENT”. In: *EURASIP Journal on Advances in Signal Processing* 2013.1 (2013), p. 145.

[9] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of model checking*. MIT press, 2008.

[10] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. “An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs”. In: *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*. IEEE. 2011, pp. 105–114.

[11] Brian Baldwin, Emanuel M Popovici, Michael Tunstall, et al. “Fault injection platform for block ciphers”. In: (2008).

[12] Hagai Bar-El et al. “The Sorcerer’s Apprentice Guide to Fault Attacks.” In: *IACR Cryptology ePrint Archive* 2004 (2004), p. 100. URL: http://dblp.uni-trier.de/db/journals/iacr/iacr2004.html\#Bar-ElCNTW04.

[13] Zuzana BaranovÃą et al. “Model Checking of C and C++ with DIVINE 4”. In: *Automated Technology for Verification and Analysis (ATVA 2017)*. Vol. 10482. LNCS. Springer, 2017, pp. 201–207.

[14] Alessandro Barenghi et al. “A fault induction technique based on voltage underfeeding with application to attacks against AES and RSA”. In: *Journal of Systems and Software* 86.7 (2013), pp. 1864–1878.

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

[16] Patrick Baudin et al. "ACSL: ANSI/ISO C Specification Language, version 1.4". In: *CEA* 6 (2009). URL: http://frama-c.com/download/acsl_1.

[17] Ray Beaulieu et al. *The SIMON and SPECK Families of Lightweight Block Ciphers*. Cryptology ePrint Archive, Report 2013/404. http://eprint.iacr.org/2013/404. 2013.

[18] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[19] Alfredo Benso et al. "Software dependability techniques validated via fault injection experiments". In: *Radiation and Its Effects on Components and Systems, 2001. 6th European Conference on*. IEEE. 2001, pp. 269–274.

[20] Maël Berthier et al. "Idea: embedded fault injection simulator on smartcard". In: *International Symposium on Engineering Secure Software and Systems*. Springer. 2014, pp. 222–229.

[21] Sebastian Biallas, Jörg Brauer, and Stefan Kowalewski. "Arcade.PLC: A Verification Platform for Programmable Logic Controllers". eng. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. ACM, 2012, pp. 338–341. ISBN: 978-1-4503-1204-2. URL: http://publications.embedded.rwth-aachen.de/file/3w.

[22] Armin Biere et al. "Bounded model checking". In: *Advances in computers* 58 (2003), pp. 117–148.

[23] Eli Biham and Adi Shamir. "Differential fault analysis of secret key cryptosystems". In: *Annual international cryptology conference*. Springer. 1997, pp. 513–525.

[24] Alex Biryukov, Arnab Roy, and Vesselin Velichkov. "Differential analysis of block ciphers SIMON and SPECK". In: *International Workshop on Fast Software Encryption*. Springer. 2014, pp. 546–570.

[25] Andrey Bogdanov et al. "PRESENT: An ultra-lightweight block cipher". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2007, pp. 450–466.

[26] Dan Boneh, Richard A DeMillo, and Richard J Lipton. "On the importance of checking computations". In: (1996).

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

[28] Jakub Breier and Wei He. "Multiple fault attack on present with a hardware trojan implementation in fpga". In: *arXiv preprint arXiv:1702.08208* (2017).

[29] Jakub Breier, Xiaolu Hou, and Yang Liu. "Fault Attacks Made Easy: Differential Fault Analysis Automation on Assembly Code". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.2 (2018), pp. 96–122.

[30] Sebanjila K Bukasa et al. "Let's shock our IoT's heart: ARMv7-M under (fault) attacks". In: *Proceedings of the 13th International Conference on Availability, Reliability and Security*. ACM. 2018, p. 33.

[31] Joao Carreira, Henrique Madeira, João Gabriel Silva, et al. "Xception: Software fault injection and monitoring in processor functional units". In: *Dependable Computing and Fault Tolerant Systems* 10 (1998), pp. 245–266.

[32] Joao Viegas Carreira, Diamantino Costa, and Joao Gabriel Silva. "Fault injection spot-checks computer system dependability". In: *IEEE Spectrum* 36.8 (1999), pp. 50–55.

[33] Boutheina Chetali and Quang-Huy Nguyen. "Industrial use of formal methods for a high-level security evaluation". In: *International Symposium on Formal Methods*. Springer. 2008, pp. 198–213.

[34] Williams Chris. "Kernel-memory-leaking Intel processor design flaw forces Linux, Windows redesign". In: (2018).

[35] Maria Christofi, Boutheina Chetali, and Louis Goubin. "Formal verification of an implementation of CRT-RSA Vigilant's algorithm". In: *PROOFS Workshop: Pre-proceedings*. 2013, p. 28.

[36] Cristina Cifuentes and Vishv M Malhotra. "Binary translation: Static, dynamic, retargetable?" In: *icsm*. 1996, pp. 340–349.

[37] Kai Cong et al. "Automatic fault injection for driver robustness testing". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM. 2015, pp. 361–372.

[38] ARM Cortex. "Cortex-M3 technical reference manual". In: *Rev. r1p1* (2006).

[39] Edward W Czeck, Daniel P Siewiorek, and Zary Z Segall. "Software-implemented fault insertion: An FTMP example". In: (1987).

[40] Fabrizio De Santis et al. "Ciphertext-only fault attacks on PRESENT". In: *International Workshop on Lightweight Cryptography for Security and Privacy*. Springer. 2014, pp. 85–108.

[41] Amine Dehbaoui et al. "Injection of transient faults using electromagnetic pulses-Practical results on a cryptographic system-." In: *IACR Cryptology EPrint Archive* 2012 (2012), p. 123.

[42] Itai Dinur. "Improved Differential Cryptanalysis of Round-Reduced Speck". In: *Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers*. Ed. by Antoine Joux and Amr M. Youssef. Vol. 8781. Lecture Notes in Computer Science. Springer, 2014, pp. 147–164. ISBN: 978-3-319-13050-7. DOI: 10.1007/978-3-319-13051-4_9. URL: https://doi.org/10.1007/978-3-319-13051-4_9.

[43] Wenliang Du and Aditya P Mathur. "Vulnerability testing of software system using fault injection". In: *Purdue University, West Lafayette, Indiana, Technique Report COAST TR* (1998), pp. 98–02.

[44] Elena Dubrova. "Fundamentals of dependability". In: *Fault-Tolerant Design*. Springer, 2013, pp. 5–20.

[45] Louis Dureuil. "Code Analysis and Certification Process of Secure Hardware against Fault Injection". Theses. Communauté Université Grenoble Alpes, Oct. 2016. URL: https://tel.archives-ouvertes.fr/tel-01403749.

[46] Louis Dureuil et al. "FISSC: A Fault Injection and Simulation Secure Collection". In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2016, pp. 3–11.

[47] Louis Dureuil et al. "From Code Review to Fault Injection Attacks: Filling the Gap Using Fault Model Inference". In: *International Conference on Smart Card Research and Advanced Applications*. Springer. 2015, pp. 107–124.

[48] Tom Durkin. "What the Media Couldn't Tell You About Mars Pathfinder". In: *Robot Science &* (1998).

[49] Robert Ecoffet. "In-flight anomalies on electronic devices". In: *Radiation Effects on Embedded Systems*. Springer, 2007, pp. 31–68.

[50] E Allen Emerson. "The beginning of model checking: A personal perspective". In: *25 Years of Model Checking*. Springer, 2008, pp. 27–45.

[51] Luis Entrena et al. "Hardware fault injection". In: *Soft Errors in Modern Electronic Systems*. Springer, 2011, pp. 141–166.

[52] Nahid Farhady Ghalaty, Bilgiday Yuce, and Patrick Schaumont. "Differential fault intensity analysis on PRESENT and LED block ciphers". In: *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer. 2015, pp. 174–188.

[53] Thomas Given-Wilson, Nisrine JAFRI, and Axel Legay. "Bridging Software-Based and Hardware-Based Fault Injection Vulnerability Detection". working paper or preprint. Dec. 2018. URL: https://hal.inria.fr/hal-01961008.

[54] Thomas Given-Wilson et al. "An Automated and Scalable Formal Process for Detecting Fault Injection Vulnerabilities in Binaries". In: (2017).

[55] Thomas Given-Wilson et al. "An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries and Case Study on PRESENT". In: *2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, Australia, August 1-4, 2017*. IEEE, 2017, pp. 293–300. ISBN: 978-1-5090-4906-6. DOI: 10.1109/Trustcom/BigDataSE/ICESS.2017.250. URL: https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.250.

[56] Enes Göktas et al. "Out of control: Overcoming control-flow integrity". In: *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE. 2014, pp. 575–589.

[57] Brian Gough. *GNU scientific library reference manual*. Network Theory Ltd., 2009.

[58] Tomás Grimm, Djones Lettnin, and Michael Hübner. "A Survey on Formal Verification Techniques for Safety-Critical Systems-on-Chip". In: *Electronics* 7.6 (2018), p. 81.

[59] Sylvain Guilley et al. "Fault injection resilience". In: *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*. IEEE. 2010, pp. 51–65.

[60] Sylvain Guilley et al. "Silicon-level solutions to counteract passive and active attacks". In: *FDTC*. IEEE-CS. 2008, pp. 3–17.

[61] Donald H Habing. "The use of lasers to simulate radiation-induced transients in semiconductor devices and circuits". In: *IEEE Transactions on Nuclear Science* 12.5 (1965), pp. 91–100.

[62] Fred H Hardie and Robert J Suhocki. "Design and use of fault simulation for Saturn computer design". In: *IEEE Transactions on Electronic Computers* 4 (1967), pp. 412–429.

[63] Niranjan Hasabnis and R Sekar. "Automatic generation of assembly to IR translators using compilers". In: *Workshop on Architectural and Microarchitectural Support for Binary Translation*. 2015.

[64] Niranjan Hasabnis and R Sekar. "Lifting assembly to intermediate representation: A novel approach leveraging compilers". In: *ACM SIGPLAN Notices* 51.4 (2016), pp. 311–324.

[65] Andrea Höller et al. "QEMU-based fault injection for a system-level analysis of software countermeasures against fault attacks". In: *Digital System Design (DSD), 2015 Euromicro Conference on*. IEEE. 2015, pp. 530–533.

[66] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*. Vol. 1003. Addison-Wesley Reading, 2004.

[67] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. "Fault Injection Techniques and Tools". In: *Computer* 30.4 (Apr. 1997), pp. 75–82. ISSN: 0018-9162. DOI: 10.1109/2.585157. URL: http://dx.doi.org/10.1109/2.585157.

[68] Mafijul Md Islam et al. "Binary-Level Fault Injection for AUTOSAR Systems (Short Paper)". In: *Dependable Computing Conference (EDCC), 2014 Tenth European*. IEEE. 2014, pp. 138–141.

[69] Nisrine Jafri, Axel Legay, and Jean-Louis Lanet. "Vulnerability Prediction Against Fault Attacks". In: *ERCIM News* (2016).

[70] Tomislav Janjusic and Krishna Kavi. "Hardware and Application Profiling Tools". In: *Advances in Computers*. Vol. 92. Elsevier, 2014, pp. 105–160.

[71] Andréas Johansson. "Software implemented fault injection used for software evaluation". In: *Building Reliable Component-Based Systems* (2002).

[72] Barry W Johnson. *Design & analysis of fault tolerant digital systems*. Addison-Wesley Longman Publishing Co., Inc., 1988.

[73] Philipp Jovanovic, Martin Kreuzer, and Ilia Polian. "A fault attack on the LED block cipher". In: *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer. 2012, pp. 120–134.

[74] Ghani A Kanawati, Nasser A Kanawati, and Jacob A Abraham. "FERRARI: A flexible software-based fault and error injection system". In: *IEEE Transactions on computers* 2 (1995), pp. 248–260.

[75] Johan Karlsson et al. "Application of three physical fault injection techniques to the experimental assessment of the MARS architecture". In: *Dependable Computing and Fault Tolerant Systems* 10 (1998), pp. 267–288.

[76] Yoongu Kim et al. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *ACM SIGARCH Computer Architecture News*. IEEE Press. 2014, pp. 361–372.

[77] Johannes Kinder and Helmut Veith. "Jakstab: A static analysis platform for binaries". In: *International Conference on Computer Aided Verification*. Springer. 2008, pp. 423–427.

[78] Johannes Kinder et al. "Proactive detection of computer worms using model checking". In: *IEEE Transactions on Dependable and Secure Computing* 7.4 (2010), pp. 424–438.

[79] Lars R Knudsen and Gregor Leander. "PRESENT–Block Cipher". In: *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 953–955.

[80] Maha Kooli and Giorgio Di Natale. "A survey on simulation-based fault injection tools for complex systems". In: *Design & Technology of Integrated Systems In Nanoscale Era (DTIS), 2014 9th IEEE International Conference On*. IEEE. 2014, pp. 1–6.

[81] Thomas Korak and Michael Hoefler. "On the effects of clock and power supply tampering on two microcontroller platforms". In: *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014 Workshop on*. IEEE. 2014, pp. 8–17.

[82] Raghavan Kumar et al. "Parametric Trojans for fault-injection attacks on cryptographic hardware". In: *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014 Workshop on*. IEEE. 2014, pp. 18–28.

[83] Marta Kwiatkowska, Gethin Norman, and David Parker. "PRISM 4.0: Verification of probabilistic real-time systems". In: *International conference on computer aided verification*. Springer. 2011, pp. 585–591.

[84] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–. ISBN: 0-7695-2102-9. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

[85] Axel Legay, Benoît Delahaye, and Saddek Bensalem. "Statistical model checking: An overview". In: *International Conference on Runtime Verification*. Springer. 2010, pp. 122–135.

[86] Axel Legay and Louis-Marie Traonouez. "Plasma Lab Statistical Model Checker: Architecture, Usage and Extension". In: *43rd International Conference on Current Trends in Theory and Practice of Computer Science*. 2017.

[87] Jacques-Louis Lions et al. *Ariane 5 flight 501 failure report by the inquiry board*. 1996.

[88] Chi-Keung Luk et al. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Acm sigplan notices*. Vol. 40. 6. ACM. 2005, pp. 190–200.

[89] Jean-Baptiste Machemie et al. "SmartCM a smart card fault injection simulator". In: *2011 IEEE International Workshop on Information Forensics and Security*. IEEE. 2011, pp. 1–6.

[90] Timothy C May and Murray H Woods. "A new physical mechanism for soft errors in dynamic memories". In: *Reliability Physics Symposium, 1978. 16th Annual*. IEEE. 1978, pp. 33–40.

[91] Tilman Mehler and Peter Leven. *Introduction to StEAM-an assembly-level software model checker*. Tech. rep. Technical Report 193, University of Dortmund and University of Freiburg, 2003.

[92] Eric Mercer and Michael Jones. "Model checking machine code with the GNU debugger". In: *International SPIN Workshop on Model Checking of Software*. Springer. 2005, pp. 251–265.

[93] Florian Merz, Stephan Falke, and Carsten Sinz. "LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR". In: *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*. VSTTE'12. Philadelphia, PA: Springer-Verlag, 2012, pp. 146–161. ISBN: 978-3-642-27704-7. DOI: 10.1007/978-3-642-27705-4_12. URL: http://dx.doi.org/10.1007/978-3-642-27705-4_12.

[94] Florent Miller et al. "Effects of beam spot size on the correlation between laser and heavy ion SEU testing". In: *IEEE transactions on nuclear science* 51.6 (2004), pp. 3708–3715.

[95] Nicolas Moro. "Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués". PhD thesis. Université Pierre et Marie Curie-Paris VI, 2014.

[96] Nicolas Moro et al. "Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller". In: *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*. IEEE. 2013, pp. 77–88.

[97] Nicolas Moro et al. "Formal verification of a software countermeasure against instruction skip attacks". In: *Journal of Cryptographic Engineering* 4.3 (2014), pp. 145–156.

[98] Mehran Mozaffari-Kermani et al. "Fault-resilient lightweight cryptographic block ciphers for secure embedded systems". In: *IEEE Embedded Systems Letters* 6.4 (2014), pp. 89–92.

[99] Jan Mrázek et al. "SymDIVINE: tool for control-explicit data-symbolic state space exploration". In: *International Symposium on Model Checking Software*. Springer. 2016, pp. 208–213.

[100] Jongwhoa Na and Dongwoo Lee. "Acceleration of Simulated Fault Injection Using a Checkpoint Forwarding Technique". In: *ETRI Journal* 39.4 (2017), pp. 605–613.

[101] Nicholas Nethercote. *Dynamic binary analysis and instrumentation*. Tech. rep. University of Cambridge, Computer Laboratory, 2004.

[102] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavy-weight dynamic binary instrumentation". In: *ACM Sigplan notices*. Vol. 42. 6. ACM. 2007, pp. 89–100.

[103] James Newsome. "Detecting and Preventing Control-flow Hijacking Attacks in Commodity Software". AAI3365680. PhD thesis. Pittsburgh, PA, USA, 2008. ISBN: 978-1-109-26104-2.

[104] Xuan Thuy Ngo et al. "Integrated sensor: a backdoor for hardware Trojan insertions?" In: *Euromicro Conference on Digital System Design (DSD) 2015*. 2015.

[105] Onur Özen et al. "Lightweight Block Ciphers Revisited: Cryptanalysis of Reduced Round PRESENT and HIGHT". In: *Information Security and Privacy, 14th Australasian Conference, ACISP 2009, Brisbane, Australia, July 1-3, 2009,*

Proceedings. Ed. by Colin Boyd and Juan Manuel González Nieto. Vol. 5594. Lecture Notes in Computer Science. Springer, 2009, pp. 90–107. ISBN: 978-3-642-02619-5. DOI: 10.1007/978-3-642-02620-1_7. URL: https://doi.org/10.1007/978-3-642-02620-1_7.

[106] Corina S. Păsăreanu and Willem Visser. "Symbolic Execution and Model Checking for Testing". In: *Hardware and Software: Verification and Testing: Third International Haifa Verification Conference, HVC 2007, Haifa, Israel, October 23-25, 2007. Proceedings*. Ed. by Karen Yorav. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 17–18. ISBN: 978-3-540-77966-7. DOI: 10.1007/978-3-540-77966-7_5. URL: http://dx.doi.org/10.1007/978-3-540-77966-7_5.

[107] Karthik Pattabiraman et al. "SymPLFIED: Symbolic program-level fault injection and error detection framework". In: *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE. 2008, pp. 472–481.

[108] Roberta Piscitelli, Shivam Bhasin, and Francesco Regazzoni. "Fault attacks, injection techniques and tools for simulation". In: *Hardware Security and Trust*. Springer, 2017, pp. 27–47.

[109] Marta Portela-Garcia et al. "A rapid fault injection approach for measuring seu sensitivity in complex processors". In: *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International*. IEEE. 2007, pp. 101–106.

[110] Marie-Laure Potet et al. "Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections". In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE. 2014, pp. 213–222.

[111] Michael J Pratt. "Introduction to ISO 10303âĂŤthe STEP standard for product data exchange". In: *Journal of Computing and Information Science in Engineering* 1.1 (2001), pp. 102–103.

[112] Charles Price. *MIPS iv instruction set*. 1995.

[113] Rui Qiao and Mark Seaborn. "A new approach for rowhammer attacks". In: *Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on*. IEEE. 2016, pp. 161–166.

[114] J-J Quisquater. "Eddy current for magnetic analysis with active sensor". In: *Proceedings of Esmart, 2002* (2002), pp. 185–194.

[115] Rochit Rajsuman. *System-on-a-chip: Design and Test*. Artech House, Inc., 2000.

[116] Thomas Reinbacher et al. "Challenges in embedded model checking-a simulator for the [mc] square model checker". In: *2008 International Symposium on Industrial Embedded Systems*. IEEE. 2008, pp. 245–248.

[117] Lionel Rivière. "Securing software implementations against fault injection attacks on embedded systems". PhD thesis. Paris: TELECOM ParisTech, 2015.

[118] Lionel Rivière et al. "A novel simulation approach for fault injection resistance evaluation on smart cards". In: *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*. IEEE. 2015, pp. 1–8.

[119] Lionel Rivière et al. "Combining High-Level and Low-Level Approaches to Evaluate Software Implementations Robustness Against Multiple Fault Injection Attacks". In: *International Symposium on Foundations and Practice of Security*. Springer. 2014, pp. 92–111.

[120] Lionel Riviere et al. "High precision fault injections on the instruction cache of ARMv7-M architectures". In: *Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on*. IEEE. 2015, pp. 62–67.

[121] Thomas Roche, Victor Lomné, and Karim Khalfallah. "Combined fault and side-channel attack on protected implementations of aes". In: *International Conference on Smart Card Research and Advanced Applications*. Springer. 2011, pp. 65–83.

[122] Frank Rogin et al. "Advanced verification by automatic property generation". In: *IET computers & digital techniques* 3.4 (2009), pp. 338–353.

[123] Cyril Roscian, Jean-Max Dutertre, and Assia Tria. "Frontside laser fault injection on cryptosystems-Application to the AES'last round". In: *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*. IEEE. 2013, pp. 119–124.

[124] Resve Saleh et al. "System-on-chip: Reuse and integration". In: *Proceedings of the IEEE* 94.6 (2006), pp. 1050–1069.

[125] Horst Benjamin Schirmeier. "Efficient fault-injection-based assessment of software-implemented hardware fault tolerance." PhD thesis. Technical University Dortmund, Germany, 2016.

[126] Bastian Schlich and Stefan Kowalewski. "[mc] square: A Model Checker for Microcontroller Code". In: *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*. IEEE. 2006, pp. 466–473.

[127] Mark Seaborn and Thomas Dullien. "Exploiting the DRAM rowhammer bug to gain kernel privileges". In: *Black Hat* (2015).

[128] Zary Segall et al. "Fiat-fault injection based automated testing environment". In: *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*. IEEE. 1995, p. 394.

[129] Carsten Sinz, Florian Merz, and Stephan Falke. "LLBMC: A Bounded Model Checker for LLVM's Intermediate Representation - (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, pp. 542–544. DOI: 10.1007/978-3-642-28756-5_44. URL: http://dx.doi.org/10.1007/978-3-642-28756-5_44.

[130] Sergei Skorobogatov. "Optical fault masking attacks". In: *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*. IEEE. 2010, pp. 23–29.

[131] Sergei Skorobogatov and Christopher Woods. "Breakthrough silicon scanning discovers backdoor in military chip". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2012, pp. 23–40.

[132] NIST-FIPS Standard. "Announcing the advanced encryption standard (AES)". In: *Federal Information Processing Standards Publication* 197 (2001), pp. 1–51.

[133] Gary Stoneburner, Alice Y Goguen, and Alexis Feringa. "Sp 800-30. risk management guide for information technology systems". In: (2002).

[134] Ting Su et al. "Combining symbolic execution and model checking for data flow testing". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 654–665.

[135] Anna Thomas and Karthik Pattabiraman. "LLFI: An intermediate code level fault injector for soft computing applications". In: *Workshop on Silicon Errors in Logic System Effects (SELSE)*. 2013.

[136] Trail of bits. *Mc-Semantics*. https://github.com/trailofbits/mcsema. 2016.

[137] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. "Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault." In: *WISTP* 6633 (2011), pp. 224–233.

[138] Harshal Tupsamudre, Shikha Bisht, and Debdeep Mukhopadhyay. "Differential fault analysis on the families of SIMON and SPECK ciphers". In: *2014*

Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). IEEE. 2014, pp. 40–48.

[139] K Umadevi and S Rajakumari. "A review on software fault injection methods and tools". In: *International Journal of Innovative Research in Computer and Communication Engineering* 3.3 (2015), pp. 1582–1587.

[140] Benjamin Vedder. "Testing Safety-Critical Systems using Fault Injection and Property-Based Testing". PhD thesis. Halmstad University Press, 2015.

[141] Ingrid Verbauwhede, Dusko Karaklajic, and Jorn-Marc Schmidt. "The fault attack jungle-a classification model to guide you". In: *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*. IEEE. 2011, pp. 3–8.

[142] Sriram Krishnamoorthy Vishal Chandra Sharma Ganesh Gopalakrishnan. "Towards Reseiliency Evaluation of Vector Programs". In: *21st IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS)*. 2016.

[143] Willem Visser et al. "Model checking programs". In: *Automated software engineering* 10.2 (2003), pp. 203–232.

[144] Gaoli Wang and Shaohui Wang. "Differential fault analysis on PRESENT key schedule". In: *Computational Intelligence and Security (CIS), 2010 International Conference on*. IEEE. 2010, pp. 362–366.

[145] Ute Wappler and Christof Fetzer. "Hardware fault injection using dynamic binary instrumentation: FITgrind". In: *Proceedings Supplemental Volume of EDCC-6* (2006).

[146] AJ Wilby and DP Neale. "Defects introduced into Metals during Fabrication and Service". In: *Materials Science and Engineering* 3 (2009), pp. 48–75.

[147] Dennis J Wilkins. "The bathtub curve and product failure behavior". In: *Reliability HotWire* 21.NOV (2002).

[148] Jim Woodcock et al. "Formal methods: Practice and experience". In: *ACM computing surveys (CSUR)* 41.4 (2009), p. 19.

[149] Satoshi Yamane, Ryosuke Konoshita, and Tomonori Kato. "Model checking of embedded assembly program based on simulation". In: *IEICE TRANSACTIONS on Information and Systems* 100.8 (2017), pp. 1819–1826.

[150] Keun Soo Yim. "The Rowhammer Attack Injection Methodology". In: *Reliable Distributed Systems (SRDS), 2016 IEEE 35th Symposium on*. IEEE. 2016, pp. 1–10.

[151] Flore Qin-Yu Yuan. "Formal framework and tools to derive efficient application-level detectors against memory corruption attacks". PhD thesis. University of Illinois at Urbana-Champaign, 2010.

[152] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. "Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation". In: *Journal of Hardware and Systems Security* (2018), pp. 1–20.

[153] Yonghua Zhu and Honghao Gao. "A novel approach to generate the property for web service verification from threat-driven model". In: *Appl. Math* 8.2 (2014), pp. 657–664.

[154] James F Ziegler and William A Lanford. "Effect of cosmic rays on computer memories". In: *Science* 206.4420 (1979), pp. 776–788.