Cross-Layer Fault Analysis for Microprocessor Architectures (CLAM)

Ihab Alshaer

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : EEATS - Electronique, Electrotechnique, Automatique, Traitement du Signal (**EEATS**)
Spécialité : Nano électronique et Nano technologies
Unité de recherche : Laboratoire de conception et d'intégration des systèmes

Analyse multi-niveaux des fautes dans les architectures de processeurs

Cross-Layer Fault Analysis for Microprocessor Architectures (CLAM)

Présentée par :

Ihab ALSHAER

Direction de thèse :

Vincent BEROULLE PROFESSEUR DES UNIVERSITES, Université Grenoble Alpes	Directeur de thèse
Paolo MAISTRI CHARGE DE RECHERCHE, Université Grenoble Alpes	Co-encadrant de thèse
Brice COLOMBIER MAITRE DE CONFERENCES, Université Jean-Monnet Saint-Étienne	Co-encadrant de thèse

Rapporteurs :

Jean-Max DUTERTRE
PROFESSEUR DES UNIVERSITES, Ecole Nationale Supérieure des Mines de Saint-Étienne
Pascal BENOIT
MAITRE DE CONFERENCES HDR, LIRMM, CNRS, Université de Montpellier

Thèse soutenue publiquement le **16 octobre 2023**, devant le jury composé de :

Vincent BEROULLE PROFESSEUR DES UNIVERSITES, Université Grenoble Alpes	Directeur de thèse
Jean-Max DUTERTRE PROFESSEUR DES UNIVERSITES, Ecole Nationale Supérieure des Mines de Saint-Étienne	Rapporteur
Pascal BENOIT MAITRE DE CONFERENCES HDR, LIRMM, CNRS, Université de Montpellier	Rapporteur
Giorgio DI NATALE DIRECTEUR DE RECHERCHE, CNRS, Université Grenoble Alpes	Examinateur
Marie-Laure POTET PROFESSEUR DES UNIVERSITES, Université Grenoble Alpes	Présidente
Athanasios PAPADIMITRIOU ASSISTANT PROFESSOR, University of the Peloponnese, Greece	Examinateur

Invités :

Christophe DELEUZE
MAITRE DE CONFERENCES, Université Grenoble Alpes

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my thesis supervisors: Prof. Vincent Beroulle, Dr. Paolo Maistri, Dr. Brice Colombier, and Dr. Christophe Deleuze. Their invaluable assistance, kindness, and unwavering support throughout these years have been instrumental in my progress. They have broadened my perspective and greatly enhanced my ability to analyze and interpret various aspects.

I would also like to extend my appreciation to my CSI members: Prof. Jean-Max Dutertre and Prof. David Hély, for generously dedicating their time, providing insightful comments, and offering valuable feedback at the start of each year of my PhD journey.

I am also grateful to Prof. Jean-Max Dutertre and Dr. Pascal Benoit for taking the time to review my manuscript. I would also like to thank Prof. Giorgio Di Natale, Prof. Marie-Laure Potet, and Dr. Athanasios Papadimitriou for agreeing to be members of the jury for my defense.

Special thanks go to my colleagues in the LCIS and TIMA labs, particularly the members of the CTSYS team in LCIS and the AMFORS team in TIMA. I am thankful to Carole, Caroline, and Karine for their assistance with administrative matters during my PhD. Also, thanks a lot to Oumayma and Gijs for their contribution to my thesis during their internships.

Last but not least, I would like to express my heartfelt appreciation to my friends here in France and in Palestine, as well as my beloved family in Palestine. To my sister Nour, and my brothers Mohammed, Mohannad, Bahaa, and Tamer, your unwavering support and encouragement mean the world to me.

This thesis has been supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) and the French National Research Agency in the framework of the "Investissements d'avenir" program (ANR-15-IDEX-02). Finally, I would like to thank Arm for giving us the opportunity to deal with RTL descriptions for real IP cores under the Arm Academic Access Agreement.

"This thesis is dedicated to the soul of my mother Aisha,
to my aunt Fatima,
and to my father Basheer"

Titre : Analyse multi-niveaux des fautes dans les architectures de processeurs

Mot clés : Sécurité matérielle, Attaques par injection de fautes, Modélisation des fautes, Simulation des fautes RTL, Jeu d'instructions de longueur variable, Analyse de vulnérabilités

Résumé : Avec l'utilisation de plus en plus répandue des systèmes embarqués, les concepteurs matériels et les développeurs de logiciels accordent une attention croissante aux problèmes de sécurité afin de protéger ces dispositifs contre les menaces potentielles. Parmi ces menaces, les attaques physiques représentent un risque important, et les attaques par injection de fautes sont parmis les méthodes les plus puissantes dans ce cadre. Cependant, une mauvaise compréhension de l'impact causé par l'injection de fautes peut conduire à proposer des contre-mesures excessives ou insuffisantes pour ces dispositifs. Cela affecte négativement le rapport performance/coût et/ou la sécurité globale du dispositif. Pour relever ce défi, des modèles de fautes réalistes sont indispensables pour comprendre les effets de l'injection de fautes. Ces modèles jouent un rôle crucial dans l'analyse des vulnérabilités potentielles des codes logiciels et des blocs matériels, ce qui permet de protéger les systèmes numériques contre de telles attaques tout en assurant un surcoût maîtrisé. Cependant, se fier uniquement à des observations empiriques de microprocesseurs où des fautes sont injectées pose des défis lors de l'inférence des modèles de fautes, limitant ainsi notre compréhension des effets causés par ces fautes.

Cette thèse présente des preuves expérimentales qui mettent en évidence les défis liés à la caractérisation et à la modélisation des effets de l'injection de fautes lorsqu'on considère un seul niveau d'abstraction du système. Pour relever cette limitation, une approche d'analyse multi-niveaux est introduite pour combler le fossé entre les études précédentes et permettre une meilleure compréhension des effets des fautes. De plus, cette thèse démontre la mise en œuvre réussie de cette méthodologie, aboutissant à l'inférence de nouveaux modèles de fautes, réalistes et précis, à la fois au niveau logiciel et matériel. De plus, l'applicabilité de ces modèles de fautes est mise en évidence pour différents programmes, cibles matérielles et techniques d'injection de fautes. Enfin, cette thèse illustre comment ces modèles de fautes peuvent être exploités pour effectuer une analyse de vulnérabilité de codes logiciels, permettant ainsi de développer des contre-mesures efficaces pour un coût maîtrisé.

Cette thèse a été réalisée dans le cadre du projet CLAM en collaboration entre le laboratoire LCIS à Valence et le laboratoire TIMA à Grenoble. Cette thèse a été supervisée par Prof. Vincent Beroulle (LCIS) et co-supervisée par Dr. Paolo Maistri (TIMA), Dr. Brice Colombier (LabHC) et Dr. Christophe Deleuze (LCIS).

Title: Cross-Layer Fault Analysis for Microprocessor Architectures (CLAM)

Keywords: Hardware security, Fault injection attacks, Fault modeling, RTL fault simulation, Variable-length instruction set, Vulnerability analysis.

Abstract: With the widespread use of embedded system devices, hardware designers and software developers started paying more attention to security issues in order to protect these devices from potential threats. Among these threats, physical attacks pose a significant risk, with fault injection attacks being a very powerful attack method. Nevertheless, an inaccurate understanding of the impact caused by fault injection can result in the proposal of either excessive or insufficient protections for these devices. This, in turn, adversely affects the performance/cost ratio and/or the overall device security. To address this challenge, realistic fault models are indispensable for comprehending the effects of fault injection. Such models play a crucial role in analyzing potential vulnerabilities in software codes and hardware designs, thereby enabling the protection of digital systems against such attacks while maintaining cost-effectiveness. However, relying solely on limited observations of faulty microprocessors poses challenges when inferring fault models, ultimately limiting our understanding of the effects caused by these faults.

This thesis presents experimental evidence that highlights the challenges in characterizing and modeling the effects of fault injection when considering a single layer of system levels. Therefore, a cross-layer analysis approach is introduced to bridge the gap between previous studies and enable a better understanding of the effects of the faults. Furthermore, the thesis demonstrates the successful implementation of this methodology, resulting in the inference of reliable and novel fault models at both software and hardware levels of abstraction. Moreover, the applicability of these fault models is showcased across various target programs, target devices, and different fault injection techniques. Finally, the thesis illustrates how these fault models can be leveraged to perform vulnerability analysis of software codes, offering the capability to develop suitable and cost-effective countermeasures.

This thesis has been performed under the CLAM project in a joint position between LCIS lab in Valence and TIMA lab in Grenoble. It has been supervised by Prof. Vincent Beroulle (LCIS) and co-supervised by Dr. Paolo Maistri (TIMA), Dr. Brice Colombier (LabHC), and Dr. Christophe Deleuze (LCIS).

TABLE OF CONTENTS

INTRODUCTION

The utilization of embedded system devices is rapidly growing across various spheres of life. For instance, according to forecasts, the number of Internet of Things (IoT) devices in use worldwide is estimated to reach approximately 30 billion by 2030 [1], as depicted in Figure 1. However, the complexity of these devices, along with their running applications, is continuously increasing. This opens the door to two considerations: The need for high performance and new methods to deal with such advances and on the other hand, the emergence of new vulnerabilities exploitable by attackers at different levels. As sensitive data are frequently processed by embedded systems, some form of protection is necessary to prevent information leakage or modification. The actual processing and protection might be vulnerable to attacks that aim at extracting this sensitive information. Physical attacks in particular are a serious threat to embedded systems.

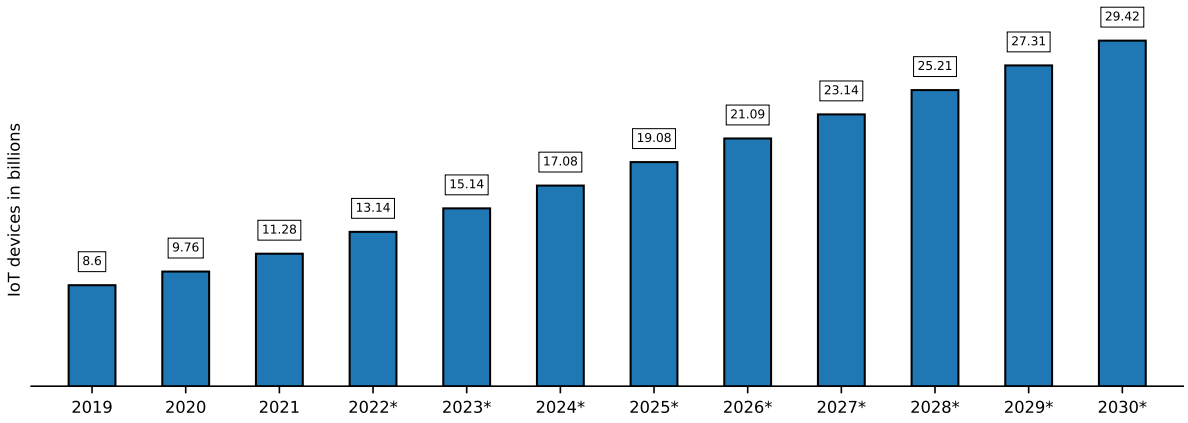


Figure 1: Number of IoT devices worldwide 2019-2021, with forecasts to 2030 (from [1]).

In the context of hardware security, physical attacks refer to various techniques and methods aimed at compromising the security of digital devices. These attacks exploit vulnerabilities in the physical properties or implementation of the device's hardware to delete, modify, gain or prevent access to confidential data.

The most prevalent physical attacks are side-channel and fault injection attacks. Side-channel attacks are passive physical attacks that primarily aim to exploit the unintentional leakage of information from a device's physical characteristics, such as power consumption [2], electromagnetic emissions [3], or timing information [4]. By capturing and analyzing these side-channel signals, attackers can infer sensitive information, such as cryptographic keys.

Fault injection attacks, on the other hand, are active physical attacks, possibly non-invasive, where the attacker will intentionally try to change the normal behavior of a device during program execution by inducing one or more faults, then observing the erroneous behavior. The resulting fault(s) could reveal an interesting behavior that could be further exploited as a vulnerability. Fault injection became an attractive research topic since the well-known Boneh *et al.* attack [5], where they were able to break some cryptographic protocols by inducing faults into the computations.

To inject a fault, a physical interference is applied on the digital device: radiations [6], laser light [7], electromagnetic pulses [8], variations of power supply [9], perturbations of clock signal [10], or changes in the environmental conditions such as the temperature [11] or else. Moreover, recent studies [12]–[14] have demonstrated the capability to perform fault injection attacks remotely by utilizing software to manipulate voltage regulators and/or energy management systems in modern devices. This has brought more attention to fault injection attacks.

In order to analyze vulnerabilities that can be exploited using fault injection attacks and propose effective countermeasures, evaluators, developers, and designers require accurate fault models. These fault models serve as abstract representations of the actual effects caused by faults, and are constructed through analysis and characterization of these effects across various levels of digital systems. It is crucial to ensure proper characterization and understanding of fault injection effects to avoid incomplete fault models. Failure to do so can result in either under-engineered or over-engineered implementations of protection measures. In the former case, security threats may persist, leaving room for exploitation, while in the latter case, unnecessary costs may be incurred, potentially leading to performance degradation.

The main objective of this thesis is to perform cross-layer analysis to examine the effects of fault injection, allowing for a better understanding of these effects at both software and hardware levels. As a result, the research will propose realistic, explainable, and trustworthy fault models across different levels of system abstraction. These

fault models will facilitate comprehensive vulnerability analysis processes and enable the effective design and development of countermeasures.

Contributions

Firstly, this thesis presents a survey of fault injection techniques that utilize timing violations to induce faults in digital systems. It also includes real-world attack examples corresponding to each technique. Additionally, the thesis examines the current state-of-the-art in terms of characterizing and modeling the effects of fault injection attacks. This analysis reveals that previous studies predominantly focused on either software or hardware aspects separately. Although some studies attempted to bridge the gap between the two levels, they only conducted simulations at the Instruction Set Architecture (ISA) and Register-Transfer Level (RTL), without validating their analysis through physical fault injections.

Furthermore, the thesis offers experimental evidence that demonstrates the limitations of characterizing and modeling fault injection effects based on a single level of analysis. As a result, the thesis proposes a comprehensive approach for cross-layer fault analysis, aiming to establish reliable fault models at various levels of abstraction. It also introduces metrics for evaluating the effectiveness of the proposed fault models.

Moreover, implementing the proposed methodology, while involving fault injections and simulations on various Arm Cortex-M processors, led to infer realistic fault models at the binary encoding of the instructions: “Skip”, “Skip and repeat”, and “non-sequential skip and repeat” for a specific number of bits. These fault models enable to explain a wide range of the obtained faulty behaviors at higher levels of abstraction, including assembly and application levels, regardless of the target instructions and target device. The provided explanations are also applicable to a diverse range of the observed faulty behaviors that have been documented in the literature. Based on the proposed models, the thesis provides exploitation and vulnerability analysis examples. This showcases the high fidelity of the proposed fault models. In addition to that, a tool to simulate these models is also presented.

Similarly, analyzing the fault effects at lower levels of abstraction led to derive reliable fault models at RTL: *anticipating the update* and *preventing the update* of a register value at a given clock cycle. By utilizing these fault models, it becomes possible to observe faults that are identical to faulty behaviors obtained by physical fault injection, or

by simulating the aforementioned binary encoding fault models. Additionally, gaining an understanding of the effects of fault injection at the RTL level led to the deduction of a novel fault model known as the *Partial update* fault model. This model offers a significant increase in the ability to explain faulty behaviors.

Finally, the thesis opens up avenues for various research directions. This encompasses utilizing the proposed fault models to assess the vulnerabilities of software based on predefined security properties. Furthermore, the thesis demonstrates the feasibility of developing cost-effective countermeasures by comprehending the potential effects of fault injection. Additionally, the thesis highlights the ability to combine faulty behaviors through the injection of multiple glitches. It also showcases the applicability of the fault models when employing different fault injection techniques.

Outline

The rest of this thesis is organized as follows: chapter 1 provides background on timing-based fault injection techniques with examples of successful attacks. Additionally, it reviews the state-of-the-art for characterization and modeling of the effects of fault injection. Experimental evidence for the need for cross-layer analysis, in addition to the proposed methodology, are demonstrated in chapter 2. Chapter 3 presents preliminary hardware fault simulation experiments. Additionally, it provides a detailed description of the inferred fault models at the binary encoding level. A detailed analysis at the hardware level and the inference of *Partial update* fault model are provided in chapter 4. Chapter 5 offers further analysis and results based on the presented work. The thesis is concluded along with future research perspectives in Conclusion and perspectives.

1

State-of-the-art

This chapter starts with describing briefly the timing constraints in a digital circuit that ensure correct operation of the circuit. It then displays the main fault injection techniques that could lead to violate these timing constraints, along with practical examples of actual attacks from the literature. These attacks show the efficiency of such injection techniques in real-life. Afterward, it reviews the state-of-the-art in terms of fault injection effects characterization and modeling. Characterizing and modeling the effects of the fault injection are significant for the sake of analyzing possible vulnerabilities of codes or designs . Moreover, this is extremely necessary to develop or design countermeasures against fault attacks. Finally, the chapter is concluded, along with comments on the presented works.

1.1 Timing constraints in a digital circuit

When designing a synchronous digital circuit, timing is of prime importance to guarantee a proper operation of the circuit. A synchronous digital circuit is a type of digital circuit that utilizes a clock signal to synchronize its components. These circuits consist of both sequential and combinational elements, which are interconnected. Combinational elements are made up of logical gates that perform digital computations, while sequential elements are memory cells known as flip-flops. Flip-flops are designed to store stateful data, which is used in the digital computations performed by the combinational elements. A standard flip-flop has an input D, and an output Q. When receiving a rising edge of a new clock cycle, a flip-flop adjusts the output Q to the value of the input D.

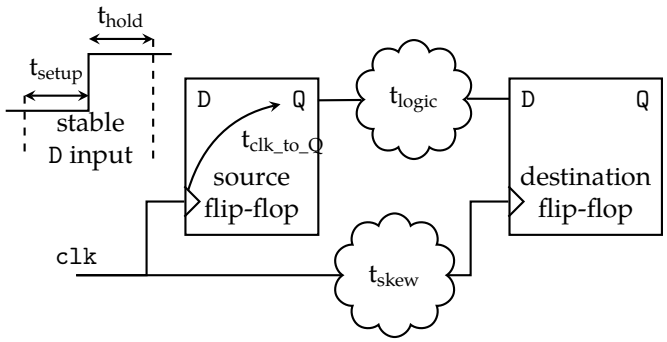


Figure 1.1: Timing metrics in a simple digital design.

The clock signal plays a crucial role in coordinating the operation of the sequential and combinational elements in synchronous digital circuits, ensuring that the computations are executed in a synchronized and orderly manner. Additionally, the clock period of a clock cycle (*i.e.,* the time between a rising edge and the next rising edge) must respect the maximum delay needed for the data to be propagated between a source flip-flop and a destination flip-flop. The path that has this maximum delay is known as the critical path in the circuit. In order to ensure correct operation of a synchronous digital design, timing must satisfy the setup and hold equations [15], [16], as presented in Equations 1.1 and 1.2 and illustrated in Figure 1.1 where:

- t_{clk} is the clock period,
- t_{setup} is the duration for which data on the *D* input must be stable before the rising edge of the clock signal [17],
- t_{hold} is the duration for which data on the *D* input must be stable after the rising edge of the clock signal [17],
- t_{logic} is the propagation delay in the combinational logic between the source and destination flip-flops,
- t_{skew} is the time difference between the arrival of the clock signal at source flip-flop and destination flip-flop,
- $t_{\text{clk\_to\_Q}}$ is the delay from the rising edge of the clock input to the *Q* output inside source flip-flop [18].

$$t_{\text{clk\_to\_Q}} + t_{\text{logic}} + t_{\text{setup}} \leq t_{\text{clk}} + t_{\text{skew}} \tag{1.1}$$

$$t_{\text{clk\_to\_Q}} + t_{\text{logic}} \geq t_{\text{hold}} + t_{\text{skew}} \tag{1.2}$$

Performing a timing-based fault injection could result in a violation of the inequalities found in one or both of Equations 1.1 and 1.2, leading to observe a faulty behavior.

1.2 Timing-based fault injection

This work primarily focuses on fault injection techniques that can lead to timing violations. Many of the currently employed and established techniques heavily depend on these violations to introduce faults in digital circuits. Moreover, these techniques offer a significant benefit in terms of cost-effectiveness when compared to alternatives like laser or radiation-based methods. Additionally, they require less expertise to successfully carry out the fault injections.

The following subsections describe the major fault injection techniques that can lead to violate the timing constraints in a synchronous digital circuit. For each outlined technique, examples of actual attacks from the literature are also given.

1.2.1 Clock glitch fault injection

Applying perturbations to the main clock signal that is fed to the digital circuit is a non-invasive and an effective fault injection technique. Clock glitch is considered as a low-cost fault injection technique compared to other techniques like laser and EM pulses. Also, it is highly controllable with respect to the temporal accuracy, and hence, the instant of the injection. This facilitates the pinpointing of specific instructions within the target program. However, since the glitch is injected in the global clock, there is no particular knowledge about which architectural element could be affected as a result of the injection. In particular, when the target device is a black box for the attacker, so has no knowledge of the device architecture. The ability of achieving the clock glitch in a remote way, as mentioned in [19], gives this kind of technique greater attention, when it comes to designing or developing protections against fault attacks. The remote attack can be accomplished using software-control of the energy management mechanisms in digital systems [19].

During a normal execution, for example, in a 3-stage pipeline, at every rising edge

of the clock, an instruction(s) is fetched by the microprocessor from the instruction memory, while another instruction (previously fetched) is being decoded or executed in another stage of the pipeline. Figure 1.2 shows a normal behavior when having a regular clock signal.

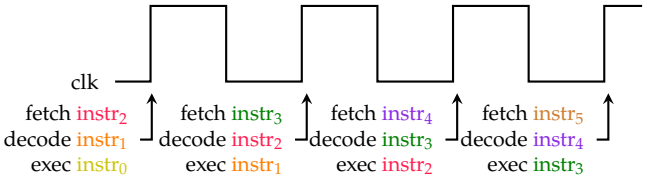


Figure 1.2: Normal behavior of a 3-stage processor pipeline with a regular clock signal.

When performing clock glitch fault injection, a glitch is injected just before or after the rising edge of the clock. This glitch would appear as a new clock cycle for the microprocessor, disrupting the regular behavior of the clock signal. Thus, resulting in a violation of the inequalities found in one or both of Equations 1.1 and 1.2, leading to various kinds of faulty behaviors.

When performing a clock glitch, three parameters must be tuned, as shown in Figure 1.3:

- Delay: the time between the rising edge of the trigger signal (used for synchronization) and the rising edge of the targeted clock cycle.
- Shift: the distance between the rising edge of the glitch and the rising edge of the targeted clock cycle.
- Width: the duration of the glitch itself.

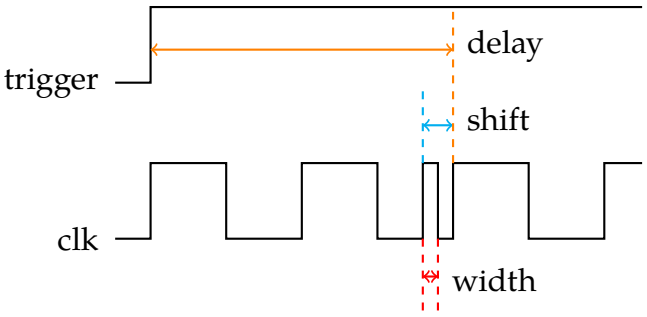


Figure 1.3: Clock glitch parameters.

It is worth mentioning that shift and width values should not be too large or too short. The too short values will not be enough to obtain a timing violation, while too large values will allow an instruction to be executed normally. Therefore, such values will result in non-observing any fault.

Examples of clock glitch fault injection attacks

Schmidt *et al.* [20] were capable of skipping a square step in the square and multiply algorithm by inducing glitches into the clock signal of an AVR microcontroller. This would facilitate breaking RSA (Rivest-Shamir-Adleman) algorithm [21]. In another work, Yuce *et al.* [10] were able to retrieve the secret key of a fault-protected implementation of the LED block cipher [22]. This is done by performing differential fault analysis attack (DFA), using clock glitch fault injection, on a LEON3 [23] processor implemented on an FPGA. DFA is a side-channel attack under the category of cryptanalysis attacks [24]. It aims at inferring secret keys by collecting several faulty outputs as a result of a fault injection campaign. Furthermore, Dobraunig *et al.* [25] succeeded in breaking different implementations of software and hardware AES (Advanced Encryption Standard) encryption algorithm [26] by carrying out clock glitch fault injection campaigns.

1.2.2 Voltage glitch fault injection

Introducing variations to the power supply that feeds a digital circuit is another effective and low-cost fault injection technique. This injection technique provides acceptable controllability in terms of temporal accuracy. However, it is hard to determine the affected part of the digital circuit as a result of the injection. As clock glitch, voltage glitch can also be performed remotely [19].

For the sake of explaining the effect of the power supply variations, or the effect of inducing glitches into the voltage that is provided to a digital circuit, several studies have been carried out. In [27]–[32], the authors showed that the propagation delay of logical gates inside a digital circuit depends on the power supply noise, in particular its average noise. Moreover, [33]–[36] showed that underpowering or introducing negative power supply glitches into a circuit led to timing violations, resulting in faulty behaviors that are identical to the ones resulting from clock glitch fault injection. The timing violation is achieved as a result of increasing the propagation delay of the logical gates, which include t_setup and t_clk_to_Q. Similarly, Zussa *et al.* [37] demonstrated

that positive power supply glitches can also lead to violate the timing constraints in the setup Equation 1.1. They explain the violation as a result of negative oscillations that are induced by the voltage pulse edges.

As an example, [34], [35] provided explanations of how the propagation delay of an inverter can be increased as a result of underpowering. Figure 1.4 shows an inverter circuit with its response propagation delay from low to high (t_{pLH}) and from high to low (t_{pHL}). The propagation delay t_{pLH} is given in Equation 1.3 [38]. Where, V_{dd} is the voltage source, C_L is the load capacitance, $V_{th,p}$ is the PMOS threshold voltage, μ_p is the holes' mobility, C_{ox} is the gate oxide capacitance, and ($\frac{W_p}{L_p}$) is the aspect ratio of the PMOS. It is obvious that decreasing the voltage source (V_{dd}) will result in increasing the propagation delay t_{pLH}. By replacing the PMOS parameters in Equation 1.3 with the corresponding ones for the NMOS (*e.g.,* $V_{th,p}$ with $V_{th,n}$) , an equation for t_{pHL} can be derived.

$$t_{\text{pLH}} = \frac{C_{\text{L}}\left[\frac{2|V_{\text{th,p}}|}{V_{\text{dd}}-|V_{\text{th,p}}|} + ln\left(3 - 4\frac{|V_{\text{th,p}}|}{V_{\text{dd}}}\right)\right]}{\mu_{\text{p}} C_{\text{ox}} \frac{W_{\text{p}}}{L_{\text{p}}}(V_{\text{dd}} - |V_{\text{th,p}}|)} \tag{1.3}$$

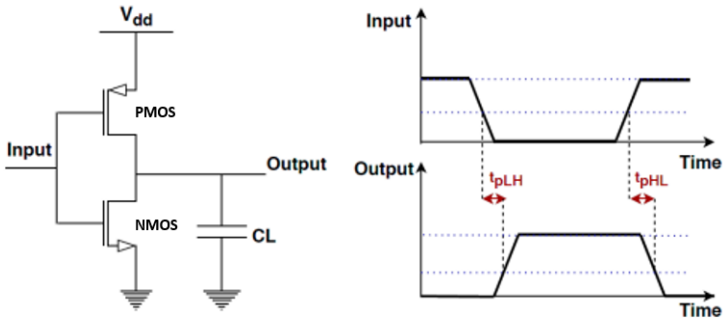


Figure 1.4: Inverter circuit with its propagation delay parameters (from [39]).

The main tuning parameters of the glitch, that need to be taken into account when performing voltage glitch fault injection, are as the following, and illustrated in Figure 1.5:

- Delay: the required time to determine the moment of injection with respect to a trigger synchronization signal.

- Length: the period of time in which the variation on the power supply will be applied.
- Amplitude: the voltage value of the pulse or the drop that is introduced to the digital circuit.

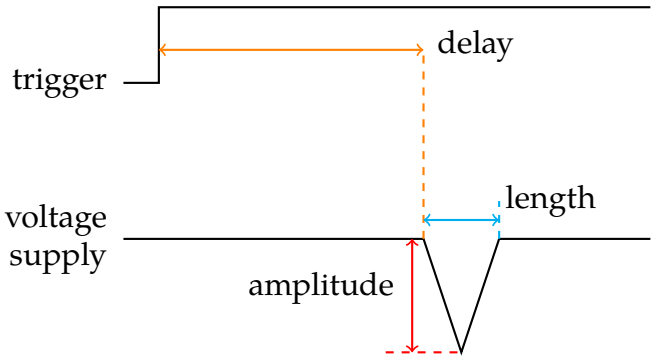


Figure 1.5: Voltage glitch parameters.

Examples of voltage glitch fault injection attacks

Bittner *et al.* [40] were able to retrieve the whole bootloader's code and decryption keys for later boot stages in a Nvidia system-on-chip, which is used in Tesla's autopilot and Mercedes-Benz's Infotainment system. Furthermore, Timmers *et al.* [9] presented voltage glitch-based attacks that led to privilege escalation in a Linux operating system, running on an Arm Cortex-A9 processor [41]. Finally, Takahashi *et al.* [42] were able to recover the full key that is used in an OpenSSL's implementation of Elliptic Curve Digital Signature Algorithm (ECDSA) [43].

1.2.3 Electromagnetic fault injection

Applying EM pulses to a digital device is an effective and a highly practical fault injection technique. This technique does not need to decapsulate the device, as the case in laser fault injection [19]. It offers better spatial accuracy than voltage and clock glitches, and roughly comparable temporal accuracy.

The EM pulses generate a magnetic field around the digital device's target part. This field may interfere with the device's normal operation. This interference may cause voltage perturbations in the circuit, leading to a rise in the propagation delay,

and hence, a setup timing violation may occur [44]–[46]. Nonetheless, other works showed that EM pulses can result in faulty behaviors that are not only explained by timing violations [47]–[49]. For example, authors in [47] illustrated that explaining EM faults as only timing violations is not enough to understand all the experimental observed faulty behaviors. This is because they were able to induce faults using EM pulses in a circuit at rest, where timing fault cannot occur.

The main parameters that need to be considered, when performing EM fault injection, are as the following:

- Probe: the coil that is used to generate the EM fields, where its physical design and characteristics play a vital role in EM fault injection [50].
- Delay: determines the time of injection with respect to a synchronized trigger signal.
- Amplitude: the voltage pulse value, which is usually produced by a pulse generator.
- Length: the amount of time in which the pulse is being generated.
- Position: determines the location of the target device with respect to the probe according to X-, Y-, and Z- axes.

Examples of EM fault injection attacks

Dehbaoui *et al.* [51] managed in recovering the encryption key of an AES software implementation by injecting a short EM pulse on a 32-bit microcontroller. Likewise, Liao *et al.* [52] succeeded in retrieving the full AES key when targeting a PIC16F687 [53]. As another example, Cui *et al.* [54] were able to perform a secure-boot violation attack on a modern multicore 1GHz Arm-based VoIP phone.

1.2.4 Heating fault injection

Changing the environmental conditions around a digital circuit, particularly the temperature, is another way to disrupt the circuit's normal behavior, which could result in observable faults. Heating a digital device may lead to an increase in the propagation delay inside it, resulting in timing violations [35], [55]. For example, in Equation 1.3, the

holes' mobility (μ_p) is directly related to the temperature. Thus, decreasing its value as a result of increasing the temperature would lead to increase t_{pLH} propagation delay [34], [55], [56]. Additionally, in [57], the authors showed that heating the target device while performing clock glitch attacks increases the observable fault rate, and hence, the success rate of the attacks.

Examples of Heating fault injection attacks

By overheating ATmega162 devices [58] beyond their maximum temperature ratings, Hutter *et al.* [59] succeeded in breaking an implementation of CRT-RSA algorithm. In another example, Govindavajhala *et al.* [60] were able to induce multiple multi-bit flips in the contents of memory chips in a desktop computer by increasing the temperature to 100°C.

1.2.5 Summary

Table 1.1 summarizes the main characteristics of each of the presented fault injection techniques in terms of: The cost of performing the injection technique, its spatial and temporal accuracy, the possibility of damaging the target device as a result of the injection, the capability of carrying out the injection remotely, and the required technical skill level to carry out the injection. The contents of this table are based on previous works in [19], [61]–[63].

	Characteristic					
Technique	Cost	Spatial accuracy	Temporal accuracy	Device damage	Remote	Technical skill
Clock glitch	low	low	ns	no	yes	moderate
Voltage glitch	low	low	ns-μs	no	yes	moderate
EM pulses	moderate	moderate	ns-μs	possible	no	moderate
Heating	low	low	none	possible	no	low

Table 1.1: Summary of the characteristics of the presented fault injection techniques.

1.3 Fault injection effect analysis and modeling

Securing digital components, such as microprocessors and microcontrollers, against fault attacks requires a thorough understanding of the faults' propagation: on the one

hand, this means characterizing, studying, and analyzing the faults that could lead to exploitable code vulnerabilities. On the other hand, it also requires designing countermeasures at different levels, hardware and software, with an acceptable cost, in terms of overhead and performance.

Fault effect analysis and characterization are performed to build the so-called "Fault models". Fault models are abstract representations of the physical fault effect. They provide description for the effects of the fault injection at different abstraction levels of a digital system. These levels refer either to software (high) or hardware (low) layers of a digital system, as shown in Figure 1.6.

Software developers and hardware designers need realistic fault models to evaluate the vulnerabilities of their codes or designs in presence of fault injection attacks. Based on this evaluation, software and/or hardware countermeasures will be developed and/or designed. On the other hand, the sufficiency of using impractical or simply random fault models will result in poor and inaccurate vulnerability analysis. Therefore, over-engineered or under-engineered countermeasures will be developed or designed. The following chapters will go into further details on these themes.

A review of the main works that employed fault injection attacks is given in the following subsections. The focus is on the level(s) at which their fault effect is described. In other words, the fault models they utilized to enable the success of their attacks and/or the fault models they proposed based on their characterization and analysis.

1.3.1 Random fault effect

This subsection presents a group of successful fault injection attacks from the literature. In all of these attacks, the employed fault models, that describe the injection effect, were *random* faults at the bit or byte level. In such attacks, the adversary has no control on the faulty bit/byte, and he/she relied on obtaining a random faulty value to allow the success of the attack.

Khelil *et al.* [33], for example, were capable of retrieving the encryption key of an AES implementation on an FPGA chip using voltage glitch fault injection. Their attack relied on faulting only one byte of the 9th round of AES and before the Mix column transformation. To do so, a reduction on the power supply of the FPGA is applied. To explain the obtained fault, they simply refer to the fact that destination flip-flops will capture bad random values as a result of increasing the propagation delay in

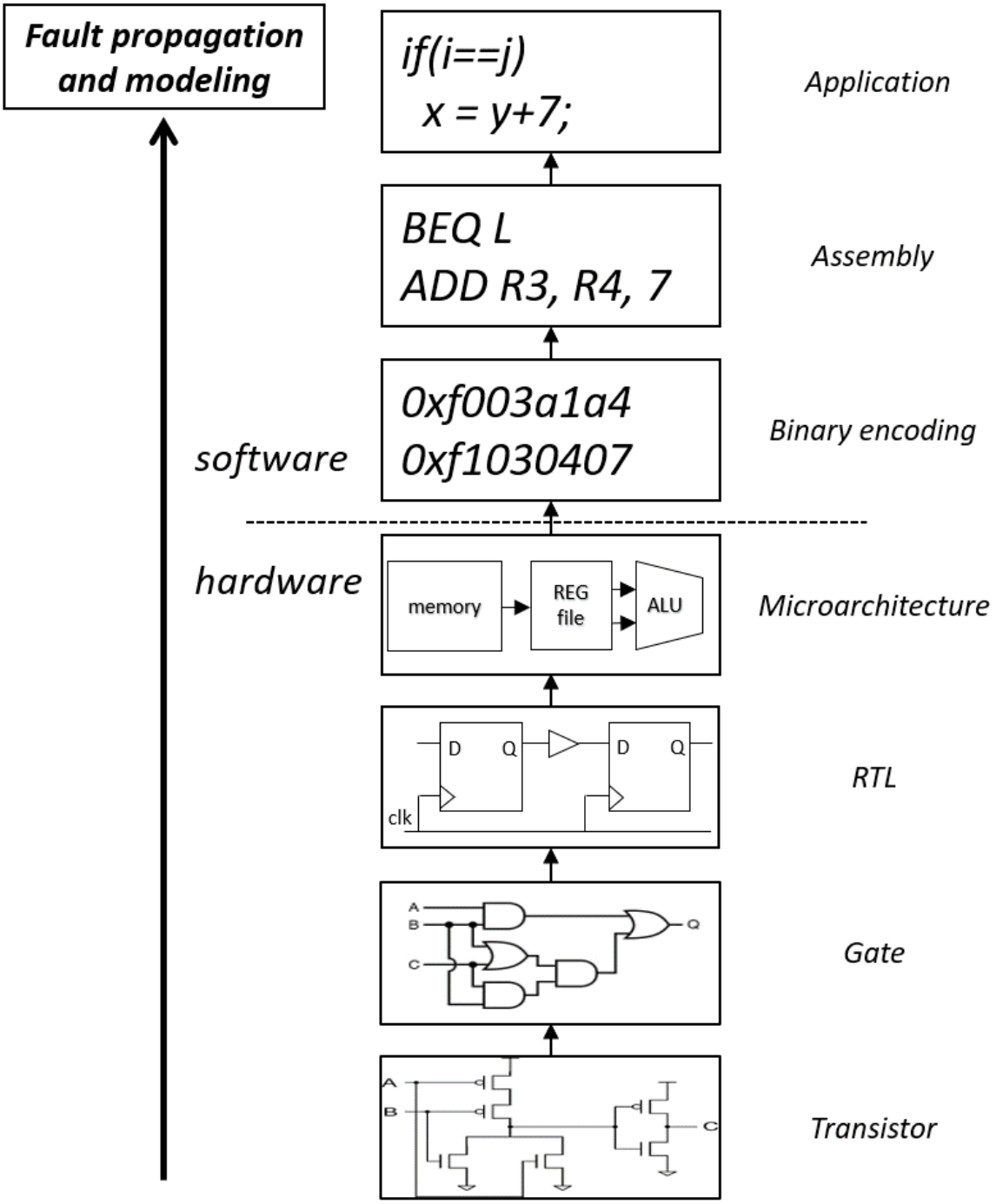


Figure 1.6: Fault propagation and modeling layers.

the computational logic between flip-flops. This increase of the delay results from the power supply reduction.

In another example, Tang *et al.* [64], succeeded in breaking the security of Arm Trustzone [65] of a Nexus6 smartphone. They were able to deduce cryptographic keys from the Trustzone. Additionally, they managed in loading self-signed applications into the Trustzone. To perform these attacks, they manipulated the clock frequency and the power supply from the voltage regulators of the Dynamic Voltage and Frequency Scaling (DVFS). DVFS is an energy-management technique that can be found in several digital devices. Such manipulation would result in violating the timing constraints of the digital elements of the target device. They described this violation with the following: an output of a source flip-flop failed in latching properly to the input of a destination flip-flop. Thus, the destination flip-flop continued to operate with stale data, which may be translated as a random bit-flip.

Similarly, Qiu *et al.* [66], exploited an implementation of the DVFS to extract the encryption key of an AES implementation that is executed in an Intel Software Guard Extensions (SGX) enclave [67]. To do so, they induced undervolting perturbations to the processor's power supply. This, as mentioned earlier, will result in violating timing constraints in the digital circuit. Thus, random faulty behaviours could occur. Their voltage glitch fault injection is totally utilized and controlled by a software that manipulates the core's voltage.

Furthermore, Chen *et al.* [68] built VoltPillager, which is a low-cost tool that allows controlling an Intel CPU voltage. They did so by exploiting a 3-wire bus called Serial Voltage Identification (SVID). SVID is responsible for transmitting the required voltage to the voltage regulator that is connected to the CPU. By injecting packets into SVID, using VoltPillager, in order to undervolting the voltage delivered to the CPU, they were able to break the confidentiality and the integrity of the CPU SGX enclaves.

Finally, Buhren *et al.* [69] succeeded in breaking the security of AMD Secure Encrypted Virtualization (SEV) [70] by carrying out voltage glitch fault injection attacks. SEV provides protections for virtual machines when used in insecure environments. As a result of their voltage glitch attack, they were able to deploy a malicious SEV firmware, which allows an attacker to decrypt the memory of the virtual machine. Additionally, they succeeded in retrieving SEV's endorsement keys, which can be used to generate fake certificates, and hence, can be used to perform software-based attacks. To perform the attack, they connect the target bus with a Teensy microcontroller [71].

Then, this microcontroller is used to program the voltage regulator of the target to apply voltage drop.

1.3.2 Fault effect analysis at ISA and microarchitecture levels

Dealing with ISA, as a result of fault injection, means involving the assembly instructions and /or their binary encoding in the characterization and analysis process. Additionally, it consists of describing the fault effect at the general-purpose registers of the processor and/or the memory locations of the target device. Several research studies have characterized faults at ISA level, due to the fact that it can be considered as the focal point for bringing high (software) and low (hardware) levels of abstraction together. On the other hand, providing description of fault effect at microarchitectural level, includes determining the faulty microarchitectural component, the pipeline stage, and/or the path inside a microcontroller or a microprocessor, where the fault is propagated. In the following, works dealt with one or both levels together are presented.

Moro *et al.* [72] performed EM fault injection campaigns on a 32-bit microcontroller that embeds an Arm Cortex-M3 processor [73]. They provided a comprehensive study to analyze the effects of EM fault injection on a real microcontroller. To explain the obtained faulty behaviors, they proposed a fault model called instruction replacement. Their approach is based on carrying out an exhaustive instruction simulation to find out which instruction can generate the same result as the observed faulty behavior. It is clear that such fault model is quite generic. Also, performing an exhaustive simulation for each faulty instruction is extremely costly. Additionally, most likely, different instructions could lead to the same result as the faulty one. Thus, uncertainty of the executed instruction would remain. With respect to fault description at microarchitectural level, they claimed that some bits, of the instruction encoding, might become faulty because of either a metastability phenomenon or the precharge value of the microcontroller's bus.

In another work, Rivière *et al.* [74] carried out practical EM fault injection experiments on an Arm Cortex-M4 processor [73]. As a result, they observed a faulty behavior where four complete 32-bit instructions are skipped, while the previous four 32-bit instructions are replayed. Based on that, they proposed a fault model where up to four 32-bit instructions or eight 16-bit instructions can be skipped, while the previous 128 bits of complete instructions are repeated. The reason behind this is the instruction

cache size, which is 128 bits. At the microarchitectural level, they described the injection effect as a fault affects the cache read at a given clock cycle, resulting in preventing the update of the cache contents. In addition to that, they provided some high level applications, where their fault model would lead to a successful attack or at least simplify an existing attack. This includes, for example: performing DFA on AES, and Bellcore [5] attack on CRT-RSA implementation.

Similar work has been presented by Trabelsi *et al.* [75]. Nonetheless, they provided additional faulty behaviors, where a combination of instructions, inside the instruction cache buffer, are affected, resulting in corrupting the value of some general purpose registers. Additionally, faults on special-purpose program status register (xPSR) are reported. In spite of that, they did not explain the rationale behind their obtained faulty behaviors. They also offered analysis at the binary encoding level of instructions, where they noticed that some faults can be explained with a single- or multi-bit reset without further explanation.

In [76], Dureuil *et al.* tried to generalize fault models as a result of performing laser and EM injections on RAMs and Flash memories of smart cards. Then, they simulated faults, based on the inferred fault models, in order to provide a so-called "vulnerability rate" for such faults. Their inferred fault models are: volatile bit set or reset on the instructions' code or data loaded from memory, non-volatile faults that affect a stored value in memory, and instruction skip. They performed their injections on two different architectures: EM injection on ARMv7-M [77], laser on a CISC machine. For evaluation purposes, they target VerifyPIN application.

In a comparable way, Werner *et al.* [78] carried out laser fault injection along with software fault simulation. However, they focused mostly on performing multi-fault attacks rather than proposing new or more thorough fault models. In particular, they aimed at combining complete instruction-skips faulty behaviors as a result of injecting multi-faults at different times. To evaluate their approach, they also target VerifyPIN application.

Moreover, Timmers *et al.* [79] were able to modify the value of the Program Counter register by targeting ARMv7-A architecture [80]. Firstly, they showed, by simulation, how this could lead to violate a secure boot. This is performed by corrupting LOAD instructions to have the Program Counter as a destination register. After that, they illustrated the possibility of doing so by corrupting LOAD instructions in a simple program composed of assembly instructions. In the best case, their success rate reached

2.7 %. In their work, they referred to the instruction corruption as bit flips over the binary encoding of the LOAD instructions.

Furthermore, Kelly *et al.* [81] illustrated a well-controlled laser fault injection campaigns on Atmel ATtiny841 8-bit microcontroller [82], which embeds an AVR processor. They focused on targeting codes that manipulate branches. As a result, they noticed different faulty behaviors that can be classified under either instruction skip, multi-registers corruption, memory corruption, or status register corruption. For the sake of vulnerability analysis, they presented areas of vulnerability in a defensive code that is used in smart cards. For instance, certified EMV (Europay, Mastercard, and Visa) cards, and a National ID card scheme.

Another work is [8], where Proy *et al.* succeeded in corrupting the iteration counter of a secure and unsecure implementations of for loops. They carried out EM fault injection campaigns on Arm Cortex-A9 processor. Based on their analysis, they provided characterization at ISA level for the observed faulty behaviors. This includes: complete instruction skip and replay, source operand substitution, control-flow hijacking (the fault breaks the control-flow integrity of the target program), and register corruption. This corruption might correspond to a reset over the most significant 16 bits of the register expected value. At the binary encoding level, they refer to the observed faults as bit flips. Nonetheless, the authors explicitly stated that some of the obtained faults remained unexplained, without a corresponding fault model.

In a different work [83], Given-Wilson *et al.* performed only software simulation on the executable binaries of the target software. This is done for the sake of providing an automated formal process to discover fault injection vulnerabilities. As fault models, they consider a modification of conditional or unconditional Jump instructions, instruction skip, and the ability to reset one or two bytes. They illustrated the efficiency of their method by detecting vulnerabilities in the PRESENT encryption algorithm [84], [85].

Back to practical fault injections, Menu *et al.* [86] conducted EM fault injection campaigns on the flash memory of an 8-bit ATmega328P microcontroller [87]. As a result, they proposed consecutive instructions skip fault model. Consequently, by considering their faut model, they succeeded in bypassing the verification of VerifyPIN application.

Moreover, Trouchkine *et al.* [88] described EM fault campaigns on two modern processors: ArmBCM2837, which embeds Cortex-A53 [89], and Intel Core i3-6100T CPU [90]. The authors also provided characterization at ISA level to propose general fault models for different architectures: one of their proposed models is random register cor-

ruption; moreover, some of their faults were still left unexplained, with unknown fault model. Other proposed fault models include instruction corruption and bit reset. With respect to evaluate the security of high-level applications based on their characterization process, they performed DFA on OpenSSL AES [91] implementation.

Finally, Khaut *et al.* [92] carried out laser fault injection campaigns on a 32-bit microcontroller that embeds an Arm Cortex-M0+ processor [93]. Based on the position of the laser shots, they obtained different faulty behaviors. This includes: complete instructions skip, and complete instructions skip and replay. At the microarchitectural level, they hypothetically refer to skip and replay faults as preventing the update of an instruction buffer with either 32-bit or 64-bit size. On the other hand, they assume that skip faults can be as a result of instructions' opcode corruption, or bit resets while the instruction is propagated from the flash memory till the execute stage in the core pipeline.

1.3.3 Fault effect analysis at lower levels of abstraction

Other studies attempted to analyze fault injection effects at lower hardware levels, such as RTL and transistor levels. This is accomplished either by focusing only on a lower level of abstraction, or by including lower levels of abstraction in the analysis along with higher levels. Following are some examples from the literature.

Fault effect analysis at RTL

For the sake of better understanding the fault propagation and bridging the analysis gap between high levels and low levels of abstraction, some studies included RTL level in the fault effect analysis process. Including RTL means trying to describe, illustrate, and show how the fault affects a single or a group of flip-flops, *i.e.,* a register or a part of it. Additionally, it shows how the fault effect is propagated between these flip-flops or registers.

Among these studies, Laurent *et al.* [94], [95], where they suggested that fault effect analysis using typical software fault models are no longer enough to characterize the observed faulty behaviors, in particular when targeting complex microprocessors that have numerous internal elements, *i.e.,* registers and combinational logic. In their work, they provided a comprehensive analysis to assess and propose new software fault models. To put into practice, they defined an approach that is based on analyzing

and comparing the results that can be obtained when performing software and RTL fault simulations on a RISC-V microprocessor [96], while executing simple assembly instructions. In terms of used fault models: at RTL, they employed bit flips fault model that is applied to a single or a few flip-flops (up to 5) within the processor pipeline. On the other hand, different versions of instruction skip, test inversion, and forwarding are employed as software fault models. Forwarding fault model corresponds to a fault that affects data forwarding (also called bypassing). Data forwarding is an optimization technique used to solve hazards whenever there are data dependencies between instructions in a program. It has been noticed that a forwarding fault could invert conditional branches. To evaluate their fault models at higher levels, they targeted VerifyPIN and LittleXorKey applications.

In a similar work, Tollec *et al.* [97] presented an automated approach of analyzing hardware and software fault simulation experiments. Their approach is based on formal verification methods by model checking. While modeling the fault effect, they explore all possible next states from a current state. They then check for states that may lead to any vulnerability that could violate a predefined security property. With respect to employed RTL fault models, they utilized bit set, bit reset, bit flip, and bit random fault. For software models, they considered instruction skip, instruction replay, incorrect order of instruction replay, and branch corruption. At application level, they evaluated their approach over four different versions of VerifyPIN application.

It should be noticed that in both previous works, physical fault injections were not performed to validate the realism of their proposed RTL and software fault models. Moreover, different architectures should be taken into account in order to generalize the assumptions of their works.

It is worth noticed that bit fault model at RTL considers disrupting input or output signals of a specific flip-flop inside a specific architectural element, which is not necessarily correspond to a bit within the binary encoding of the instructions' data, for example. It may correspond, for instance, to a control signal as described in [94].

Fault effect analysis at Transistor level

Other research works provided characterization for the fault injection effects at Transistor level. Two of these studies are presented in the following.

Dutertre *et al.* [98] conducted laser fault injection campaigns on a specific test chip that consists of CMOS 28 nm technology node, where special D flip-flops were

designed. These flip-flops were assembled in an in-line shift register composed of 10 D flip-flops, and a matrix-shaped shift register composed of 64 D flip-flops. While fault injection, they were able to identify sensitive areas of the transistors that form these flip-flops, where single-bit set/reset or flip faults could be obtained. Additionally, more faulty bits were achievable by increasing the laser pulse energy. At a higher level, they evaluated their analysis on a hardware AES encryption unit embedded on the same chip test.

In another study, Colombier *et al.* [7] carried out laser fault injection campaigns on a 32-bit microcontroller that embeds an Arm Cortex-M3 processor. They provided a comprehensive analysis of how the laser shots could affect connected transistors in parallel between a bit-line and the ground. By exploiting these bit lines, they were able to induce a single bit set or two adjacent bit sets to the binary encoding of the instructions while being fetched from the flash memory. As a result, corruption over either the operands or the opcode of the instruction was observed. As a propagation effect, control-flow corruption was achieved. For high level applications, they performed attacks over an implementation of VerifyPIN, and an implementation of AES encryption algorithm.

1.3.4 Summary

To summarize, this section begun by presenting recent fault injection attacks from the literature, which primarily focused on achieving successful fault injection. However, it is worth noting that these attacks typically refer to the resulting fault effect either as random bit or byte faults. In contrast, subsection 1.3.2 and subsection 1.3.3 provided a more detailed characterization of the fault effects and proposed fault models based on the analysis. Obviously, these works focused either on a single level of analysis or on simulation. As a result, they provided quite generic fault models that would make the vulnerability analysis harder. Consequently, insecure and/or costly countermeasures will be designed or developed. Table 1.2 summarizes the presented works, in subsection 1.3.2 and subsection 1.3.3, in terms of the following: the injection type (the employed injection technique and/or the level of the implemented simulation; software or hardware), the target processor or embedded architecture, the high-level target application, the used and/or proposed fault models at ISA, binary encoding of instructions or memory data, and RTL levels. This table clearly indicates that no prior work has

provided a comprehensive cross-layer analysis that combines hardware and software levels while conducting physical fault injection experiments.

Reference	Injection type	Target	Application	ISA	Binary	RTL
Moro *et al.* [72]	EM, SW simulation	Cortex-M3	none	instruction replacement	mulit-bit fault	none
Rivière *et al.* [74]	EM	Cortex-M4	AES, CRT-RSA	multi instructions skip and replay	none	none
Trabelsi *et al.* [75]	EM	Cortex-M4	none	register, xPSR corruption, opcode operand substitution	multi-bit reset	none
Dureuil *et al.* [76]	EM, laser, SW simulation	ARMv7-M, CISC	VerifyPIN	instruction skip, (non)-volatile memory fault	bit (re)set	none
Werner *et al.* [78]	Laser, SW simulation	Cortex-M4	VerifyPIN	multi-instrucion skip	none	none
Timmers *et al.* [79]	Voltage glitch, SW simulation	ARMv7-A	secure-boot	instruction corruption	bit flips	none
Kelly *et al.* [81]	Laser	AVR	defensive code in smartcards	instruction skip, memory & registers corruption	none	none
Proy *et al.* [8]	EM	Cortex-A9	for loops	instruction skip and replay, register corruption, operand substitution, control-flow hijacking	bit flips	none
Given-Wilson *et al.* [83]	SW simulation	none	PRESENT	jump modification, Instruction skip	1 or 2 bytes reset	none
Menu *et al.* [86]	EM	AVR	VerifyPIN	skip consecutive instructions	none	none
Trouchkine *et al.* [88]	EM	Cortex-A53, i3-6100T	OpenSSL AES	register corruption, instruction corruption	bit reset	none
Khaut *et al.* [92]	Laser	Cortex-M0+	none	instructions skip, skip and repaly	bit resets	none
Laurent *et al.* [94], [95]	HW simulation, SW simulation	RISC-V	VerifyPIN LittleXorKey	instruction skip, test inversion, forwarding	none	bit flips
Tollec *et al.* [97]	HW simulation SW simulation	RISC-V	VerifyPIN	Instruction skip, replay incorrect order of replay, forwarding, branch corruption	none	bit reset, set, flip, random
Dutertre *et al.* [98]	Laser	ASIC	hardware AES	none	none	bit(s) set, reset, flip
Colombier *et al.* [7]	Laser	Cortex-M3	VerifyPIN AES	instruction corruption, control-flow corruption	single bit set, two adjacent bit sets	none

Table 1.2: Summary for the state-of-the-art of fault injection effect characterization and modeling.

1.4 Conclusion

To sum up, this chapter presented the timing constraints in a simple digital design. Then, it demonstrated different fault injection techniques that could lead to violating

these constraints, resulting in various faulty behaviors. Several given examples demonstrated how these resulting faulty behaviors could be used to carry out harmful attacks.

After that, this chapter reviewed the state-of-the-art in terms of fault injection effect characterization, analysis, and modeling. It was shown that many works described the effect of the fault injection as random bit or byte faults. Conversely, several studies concentrated on characterizing the fault injection effect only at a single abstraction level, specifically at the ISA level. Nonetheless, some of these works tried to provide additional analysis at the microarchitectural level by considering the faulty architectural component or fault propagation path. However, all of these studies proposed rather broad fault models, such as instruction skip and instruction corruption. These fault models are clearly not enough to evaluate vulnerabilities of software codes or hardware designs. Additionally, they would lead to develop or design non-optimal countermeasures. This certainly will affect either the cost, the performance or the security of the device. A few studies, on the other hand, combined high and low levels of abstraction in an effort to promote comprehension of the effect of fault propagation in a digital system. However, they only carried out fault simulations on a RISC-V processor and no physical injections at all were performed. By incorporating physical fault injections, the realism, and reliability of the proposed models can be better assessed, and the findings can be more applicable to a wider range of system architectures. Finally, other works attempted to focus their analysis at the lowest level of abstraction by describing how the induced fault affects the normal behavior of transistors.

2

The need for cross-layer analysis and proposed methodology

In the context of the increasing complexity in embedded microprocessors and the associated behaviors in the presence of fault injection attacks, the need for realistic fault models becomes crucial for studying code vulnerabilities and protecting digital systems from these attacks. However, deriving accurate fault models based on limited observations of faulty microprocessors poses significant challenges. In this chapter, we present experimental evidence that highlights the difficulty of characterizing and modeling fault injection effects, in particular when only focusing on a single abstraction level for analysis. From there, we propose a holistic approach for fault analysis that encompasses different abstraction levels of a digital system, aiming to develop comprehensive fault models. This proposed methodology will definitely ease the process of vulnerability analysis and facilitate the design of effective countermeasures at a reasonable cost. Finally, we provide metrics to evaluate the quality of fault models, which can aid in assessing the accuracy and reliability of the proposed models.

2.1 Experimental evidence

Previous works usually focus on one single abstraction level of a digital system for fault effect analysis, and model the faulty behaviors at just one level. This section highlights the strong need of addressing several abstraction layers at the same time in order to fully understand the fault occurrence mechanisms. To achieve this, physical fault injection experiments are conducted on various target codes and devices. These

experiments aim to determine if the obtained faulty behaviors can be consistently characterized when making modifications to the target codes or other parts of the program. Additionally, the experiments are carried out to assess if the observed behaviors differ among different target devices.

This section presents the experimental setup, the results of the experiments and the related discussion that followed.

2.1.1 Experimental Setup

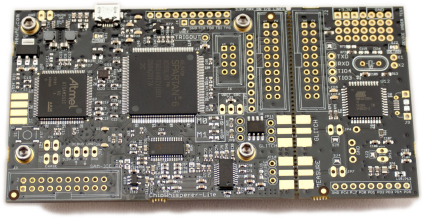
Clock glitch fault injection campaigns have been performed as physical fault injection. As mentioned in subsection 1.2.1, it is a low-cost, non-invasive, and effective fault injection technique. In the following, the target devices, the target programs, and the injection parameters are presented.

Target devices

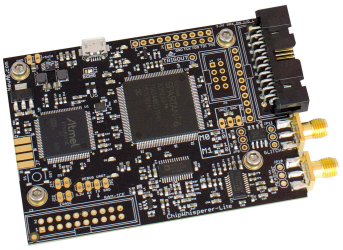
The boards that are used for the experiments are the ChipWhisperer [99] boards: CW1173 ChipWhisperer-Lite (Figure 2.1a), ChipWhisperer-Lite Capture (Figure 2.1b), and CW308 UFO baseboard (Figure 2.1c) with different targets (*e.g.*, Figure 2.1d). These ChipWhisperer boards have a dedicated environment for side channel analysis, voltage, and clock glitch of the target Arm core. We will leverage the clock glitch capabilities of this setup in the experiments. During an experiment, the ChipWhisperer-Lite or ChipWhisperer-Lite Capture board is connected to a control computer through a USB cable.

The targets are 32-bit microcontrollers, each of them embeds one of the Arm Cortex-M cores: STM32F0 embeds Cortex-M0 [100], STM32F1 embeds Cortex-M3 [73], and STM32F3 embeds Cortex-M4 [73]. STM32F3 is the connected target in CW1173 ChipWhisperer-Lite board, as shown in Figure 2.1a. The Cortex-M0 core supports the Thumb-1 instruction set [101] and a small part of the Thumb-2 instruction set [102], while Cortex-M3 and Cortex-M4 cores support Thumb-2 entirely.

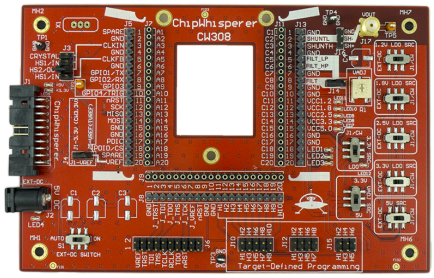
Each of these Arm cores has 13 general-purpose 32-bit registers; `R0` to `R12`. The cores also include a pipeline with three stages: fetch, decode and execute. Additionally, Cortex-M3 and Cortex-M4 have a hardware integer divide. Cortex-M4 has also additional components compared to the others, for example, it has a floating point unit and a digital signal processing unit.



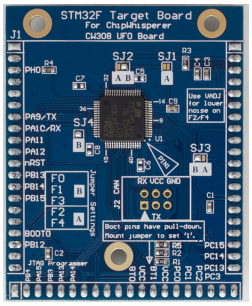
(a) CW1173 ChipWhisperer-Lite board



(b) ChipWhisperer-Lite Capture board



(c) CW308 UFO baseboard



(d) CW308 target board

Figure 2.1: ChipWhisperer boards used in the experiments.

Target programs

The injection is performed directly into inline assembly instructions within a C program in order to provide fault effect characterization at ISA level. In order to better analyze the process of the injection, the program is divided into three parts as follows:

- Prologue: instructions for the initialization and putting the processor in a known state before the injection happens.
- Target: instructions targeted by the fault injection, as well as extra instructions that would allow observing any propagation effect.
- Epilogue: instructions for reading general purpose registers [`R0-R12`] and Application Program Status Register (`APSR` *i.e.*, `Negative` (N), `Zero` (Z), `Carry` (C) and `Overflow` (V) flags); the values are transferred through serial communication commands to the control computer.

Two series of `NOP` instruction are used to isolate the three parts. This is done to ease the process of the injection by limiting the search space of the injection parameters, especially the delay. This also ensured that the prologue and the epilogue are not

affected by the injection. The `NOPs` were not deleted after compilation, as no optimization option is used for the compiler.

In the injection campaigns, two programs are targeted as shown in Listing 2.1 and Listing 2.2. Specific instructions in the target part of these programs are used to allow observing faulty effects on the control- and/or the data-flow of the program, respectively: any real-life application can be described in terms of its data flow and/or its control flow.

The use of these instructions also allows for observing other things. Firstly, it shows if the resulting faulty behaviors are related to these instructions or not, hence giving a better understanding of what triggers the faulty behavior. Secondly, it allows to check if we will be able to reproduce some faulty behaviors that are already mentioned in the literature. It also aims at obtaining the possible faulty behaviors based on the program flow, either the control flow as in Listing 2.1 or the data flow as in Listing 2.2. This includes, for example, a fault that could break the control-flow integrity, or a fault that could propagate to later instructions, and thus, breaks the data-flow integrity. Finally, it helps to understand if software characterization at the ISA level is sufficient to build realistic fault models based on the observations.

```
1 CMP R4, R6        // r4-r6 then updates APSR
2 BNE labelx        // if (Z!=1): jump to labelx
3 ADD R2, R4, R6    // r2 = r4 + r6
4 labelx:
5 ADD R5, R4, R6    // r5 = r4 + r6
```

Listing 2.1: Target part in *Program 1*: target control flow.

```
1 ADD R1, R1, 0x6   // r1 = r1 + 0x6
2 ADD R3, R3, 0xA   // r3 = r3 + 0xA
3 ADD R4, R4, 0xB   // r4 = r4 + 0xB
4 ADD R5, R6, R3    // r5 = r6 + r3
5 ADD R3, R3, 0xF   // r3 = r3 + 0xF
```

Listing 2.2: Target part in *Program 2*: target data flow and arithmetic operations.

Our goal is not to provide a complete characterization of every possible instruction, but rather provide a simple yet efficient approach that will cover as much as possible the target architecture, and emphasize diverging behaviors due to fault occurrences. For

this reason, we used instructions that explicitly have effects on different architectural elements, such as the `APSR` flags and the arithmetic logic unit. For the rest of this chapter, we will refer to the first target program in Listing 2.1 as *Program 1* and to the second target program in Listing 2.2 as *Program 2*.

For *Program 1*, the glitch is injected at the beginning of the target part. The remaining instructions aim at observing possible propagation effects. The registers `R4` and `R6` used in the experiments were initialized in the prologue to different values. Therefore, in a golden run, the `Zero` flag remains clear, the branch is taken, and the instruction at line three is not executed. We use the term "golden" to refer to the normal behavior of a program execution (*i.e.*, without any injection).

For *Program 2*, just as *Program 1*, the glitch is injected to affect the beginning of the target part. However, we distinguish two cases of execution, as two different delay values are used for the glitch. This was undertaken because different kind of faulty behaviors were observed on the targeted instructions when using another delay value, aimed at the execution of the `NOP` series before the target part.

Injection parameters

The injection campaigns consist in repeating the clock glitch fault injection 10 000 times for the same shift, width, and delay parameters. A single glitch is injected during each program execution. In the performed experiments, the glitch parameters were tuned to obtain successful faults and trying to maximize the number of the observed faults for the instructions at the beginning of the target part of each program. Nevertheless, no exhaustive search of parameters has been followed. The parameters' values are given here for reference, but it is important to emphasize that they can change according to the target device and the target program that are used in the experiment, or even environmental conditions such as temperature.

Table 2.1 shows the shift and the width values that are used for each target device (these parameters have been presented in subsection 1.2.1). The values are expressed in percentage of one clock period. The negative value of the shift means that the glitch is injected before the rising edge of the targeted clock cycle. With respect to the delay parameter, different factors can affect its value: the starting point of the trigger, the number of instructions in the prologue, the number of `NOPs` between the prologue and the target, and the position of the target instruction in the target part.

Device \ Parameter	**Width**	**Shift**
Cortex-M0	16	-14
Cortex-M3	10	-12
Cortex-M4	10	-12

Table 2.1: Glitch width and shift values used in the fault injection campaigns experiments (values in % of clock period).

2.1.2 Experimental results

This subsection presents the results of the performed experiments, and it also describes the obtained faulty behaviors for the different used target devices. Three cases can occur as a result of the fault injection, regardless of the target program, with respect to a golden (reference) behavior as follows:

- Silent: this corresponds to the case when the outcome of the injection is identical to the golden state.
- Fault: this happens when the outcome state is different from the golden one.
- Crash: this class contains the cases when the fault injection causes a crash, a reset, or a failure when getting the final state of the target through the serial channel.

The rest of this subsection is organized as follows: firstly, it presents the results for *Program 1*, secondly, it presents the outcomes for *Program 2*. The detailed analyses of the obtained results are discussed in subsection 2.1.3.

Program 1 results: Control flow target

The results of the injection campaigns for the different target devices with regard to the three categories are shown in Table 2.2. Cortex-M4 target device has the most successful faults, while Cortex-M3 has the most silent cases and Cortex-M0 has the most crashes. The obtained faulty behaviors for the different devices are described in Figure 2.2. The x-axis presents the different observed faulty behaviors, while the y-axis shows their percentages out of the successful faults, *i.e.* without Crash and Silent cases.

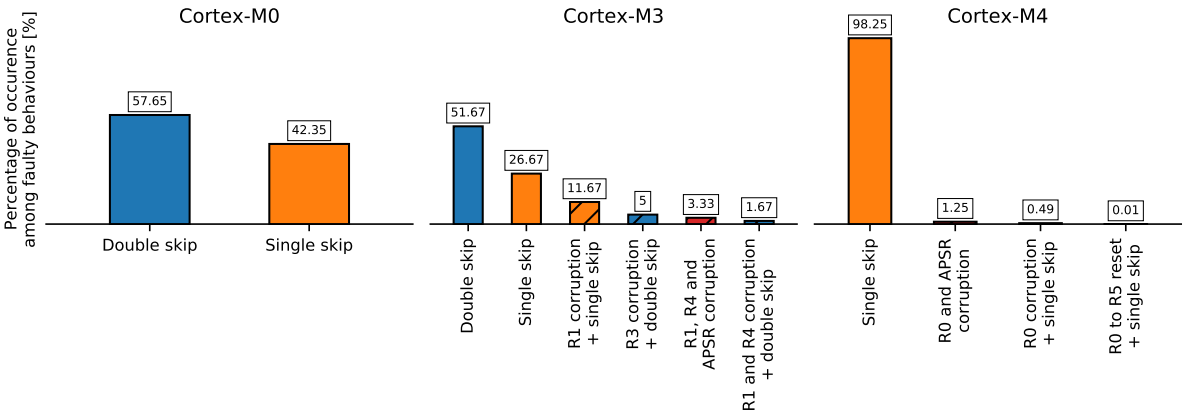


Figure 2.2: Observed faults for *Program 1* for all target devices.

Complex faulty behaviors appeared as a combination of simpler faults, even if we only performed single fault injections. For example, the result of a single fault could be an instruction skip and corruption of `R0` at the same time.

Device \ Case	**Silent**	**Crash**	**Fault**
Cortex-M0	44.08	35.66	20.26
Cortex-M3	97.76	1.64	0.60
Cortex-M4	0.01	1.18	98.81

Table 2.2: Percentage of classification cases when performing clock glitch fault injection on each target device running *Program 1*.

During these campaigns, the following faults have been observed:

- Skip: it can be either a single or a double skip. In other words, either we skip only the `CMP` instruction at line one in Listing 2.1, only the `BNE` instruction at line two, or both. If `APSR` flags have not been updated, then we assume that the `CMP` instruction was skipped. If `APSR` flags have been updated correctly and the `ADD` instruction at line three is executed, then we assume that the `BNE` instruction was skipped. If `APSR` flags have not been updated and the `ADD` instruction at line three is executed, then we assume that both instructions were skipped.

- `R0` corruption: the value of `R0` is different from its golden value. Among these corrupted values, we noticed the following: 0 (*i.e.*, the value of `R0` becomes 0), right shift by 8, 16 or 24 bits.

- `R1` Reset: `R1` value becomes 0.
- `R3` Reset: `R3` value becomes 0.
- `R4` corruption: either reset or right shift by 1 bit.
- `R0-R5` Reset: all the values of `R0` to `R5` become 0.
- `APSR` corruption: one or more of `APSR` flags have different values from the golden ones.
- Propagation effect on `R2`: it is caused by executing the `ADD` instruction at line three. The execution of this instruction can be explained as the consequence of two events. The first explanation is that the `BNE` instruction at line two in Listing 2.1 was skipped. The second explanation is that the `Zero` flag was corrupted. This leads to the branch not being taken as in a normal case, where the `Zero` flag is 0. Instead, as a result of the injection, the `Zero` flag was set to 1. These two cases could not be discriminated, as both of them might even occur together. In this experiment, this behavior only appeared in Cortex-M0 and Cortex-M3, but not Cortex-M4.
- Propagation effect on `R5`: as a result of the corrupted value in `R4`, `R5` has a wrong value at the end, since it is the sum of `R4` and `R6`.

A second experiment has been carried out with the same fault injection parameters (*i.e.*, shift, width and delay) and initialization values but with a duplicated `CMP` instruction as shown in Listing 2.3. The second experiment has been performed to see if the faulty behaviors were consistent and to improve the understanding of the induced errors. In particular, its objective was to gain insight about the reason for the propagation effect on `R2` as described above.

Regarding the three cases, Table 2.3 shows their percentages after this experiment. We can see that more faults were observed for Cortex-M0 in the second experiment, while no crash cases were obtained. Regarding the Cortex-M4, there was a significant increase in the crash category and a decrease in the successful faults. For Cortex-M3, both experiments were comparable in terms of population.

The results are shown in Figure 2.3. In addition to skip, `APSR` corruption and propagation effect on `R5`, the following behaviors were observed:

```
1 CMP R4, R6
2 CMP R4, R6
3 BNE labelx
4 ADD R2, R4, R6
5 labelx:
6 ADD R5, R4, R6
```

Listing 2.3: Target part in *Program 1* in the second experiment with duplicated `CMP`.

Board \ Case	Silent	Crash	Fault
Cortex-M0	61.29	0.0	38.71
Cortex-M3	96.07	2.69	1.24
Cortex-M4	0.88	39.32	59.80

Table 2.3: Percentage of classification cases when performing clock glitch fault injection on each target device running *Program 1 with duplicated CMP*.

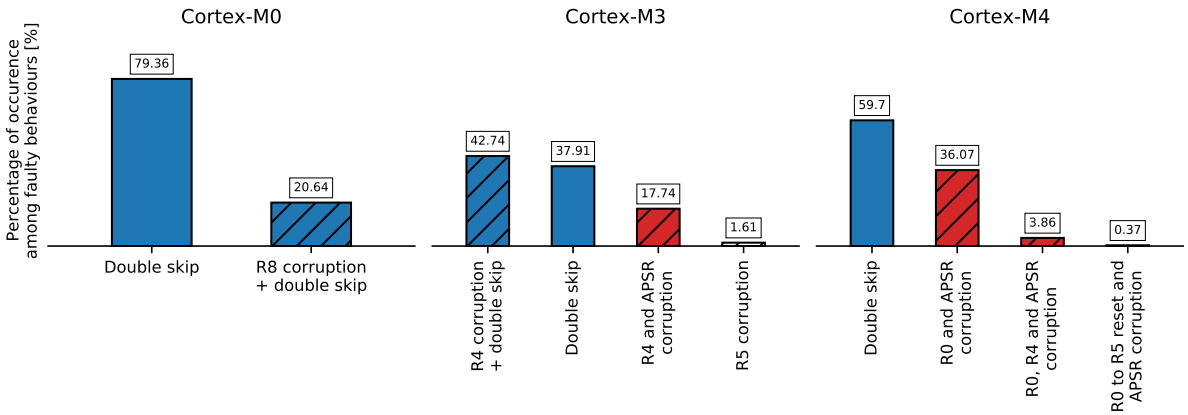


Figure 2.3: Observed faults for *Program 1 with duplicated CMP* after the second experiment for all target devices.

- `R0` Reset.
- `R4` corruption: different faulty values appeared in `R4`: 0, left shift of `R6` value by 10 or 14 bits, and another faulty value that is equal to `R7`.
- `R5` Corruption: either `R5` has its initial value or the value of `R7`. Having the initial value can be considered as a single skip as well. This could happen due to a fault while fetching this instruction and executing the previous ones.

- `R8` corruption: either set (*i.e.*, every bit has 1) or the value of `R2`.
- Propagation effect on `R2`: since we target only the beginning of the target instructions, this can not be caused by a skip or other perturbation of the `BNE` instruction. Therefore, this is necessarily caused by corruption of the `Zero` flag. This time, this behavior only appeared in Cortex-M3.

***Program 2* results: Data flow and arithmetic operations target**

The results of the three categories for this experiment are shown in Table 2.4. Again, almost all the injections resulted in successful faults in the Cortex-M4 device, while they were silent in the Cortex-M3 device. The obtained faulty behaviors are presented in Figure 2.4.

Board \ Case	**Silent**	**Crash**	**Fault**
Cortex-M0	75.51	17.28	7.21
Cortex-M3	97.93	1.2	0.87
Cortex-M4	0.0	1.16	98.84

Table 2.4: Percentage of classification cases when performing clock glitch fault injection on each target board running *Program 2*.

A wide range of faulty behaviors is observed after this experiment as the following:

- Skip: it can only be skipping the first instruction, only the second, only the third, both the first and the third, both the second and the third, or the first 4 instructions (*i.e.* quad skip).
- Repeat: repeat the first instruction. This behavior appeared as a combination with skipping the second and the third instructions. In this experiment, it is only observed for the Cortex-M0 target device.
- `R0` corruption: different faulty values observed in `R0`: set, reset, right shift of its original value by 4 or 20 bits, left shift of `R2`, *etc.*
- `R1` corruption: Among the faulty values, there were: reset, a value that is related to the program counter, left shift of `R2`, the sum of `R3` and `0x6` instead of `R1` and `0x6`, *etc.*

- R2 corruption: set only the most significant bit of the 32 bits. It only appeared for Cortex-M4 device.

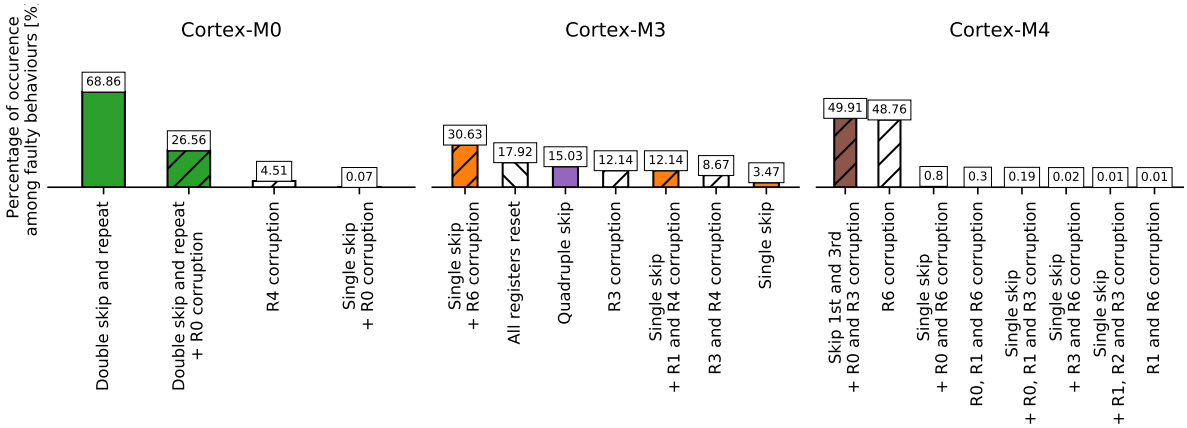


Figure 2.4: Observed faults for *Program 2* for all target boards.

- R3 corruption: faulty value related to the program counter, left shift of the original value of R1, the sum of the initial of R3, 0x6 and 0xF instead of R3, 0xA and 0xF, and other faulty values with no obvious relation.
- R4 corruption: between the faulty values that are found: the sum of R0 and 0xB, R1 and 0xB, R2 and 0xB or R3 and 0xB instead of R4 and 0xB.
- Propagation effect on R5: as a result of a faulty value that is found either in R3 or R6.
- R6 corruption: reset, the most significant bit is only set, the value of R0, left shift of R0 by 4 bits, left shift of R2, *etc*.
- All registers reset: it is only observed for Cortex-M3 target device.

A second experiment has been carried out for *Program 2*, but with adding only a NOP instruction to the prologue part. The main objective of this experiment was to investigate the consequences of a simple modification in the prologue to the target part of the program. Identical injection parameters were used, except adjusting one of the delay values. This is done to take into consideration the instruction that is added to the prologue. Table 2.5 shows the percentages of the obtained results classes. We can see a significant decrease in the successful faults for Cortex-M4 device, while similar

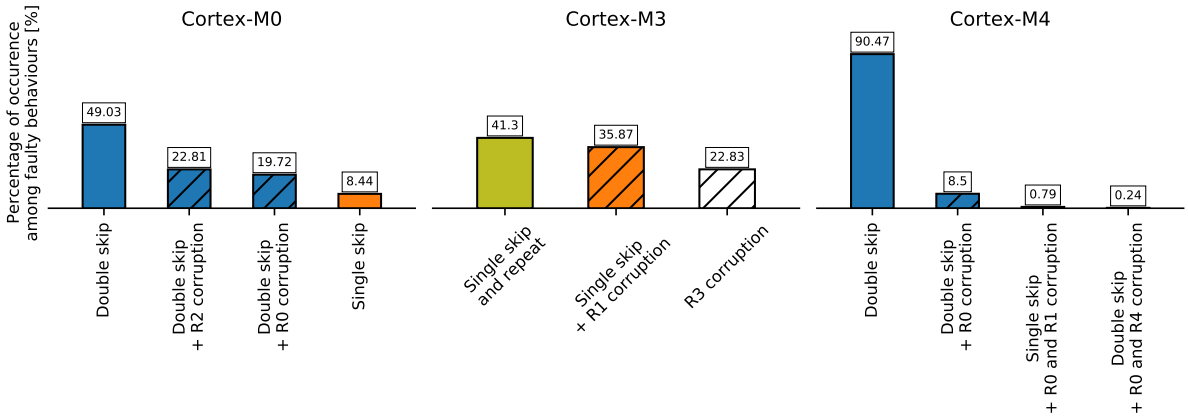


Figure 2.5: Observed faults for *Program 2 with additional NOP* for all target devices after the second experiment.

proportions for other targets with respect to the first experiment. The observed faulty behaviors are shown in Figure 2.5.

Board \ Case	**Silent**	**Crash**	**Fault**
Cortex-M0	75.07	12.78	12.15
Cortex-M3	98.55	0.99	0.46
Cortex-M4	49.85	0.24	49.91

Table 2.5: Percentage of classification cases when performing clock glitch fault injection on each target board running *Program 2 with additional NOP*.

In addition to the propagation effect on `R5`, the following faults have been observed:

- Skip: single skip only appeared for the second instruction, while double skip only occurred for the first and the second instruction.
- Repeat: again, the first instruction is repeated. This time, it is obtained as a combination with skipping only the second instruction. This behavior only appeared for the Cortex-M3 device.
- `R0` corruption: only the following cases are observed this time: set, reset, only the most significant bit is set.
- `R1` corruption: reset, a value related to the program counter, the sum of `R0` and `0x6` and other faulty values without an obvious relation.

- `R2` corruption: set, reset and other large values. This behavior only appeared for the Cortex-M0 device.
- `R3` Corruption: `R3` has a seemingly random value. It only appeared for the Cortex-M3 device.
- `R4` corruption: the sum of `0xB` and setting the most significant bit. It is only observed for the Cortex-M4 device.

2.1.3 Discussion

The aforementioned experimental results led to various conclusions, observations, and questions. The following subsections discuss the results in details with respect to different aspects.

Target device dependency

For the same target program, different faulty behaviors can be observed depending on the target device that is used in the experiment. For example, in *Program 1* experiments, `R8 corruption` is only observed for the Cortex-M0 device, while `R0 to R5 reset` is only observed for the Cortex-M4 device. Another example, in *Program 2* experiments, `all registers reset` behavior is obtained only for the Cortex-M3 device, while `skip and repeat` faulty behavior is observed for the Cortex-M0 and the Cortex-M3 but not for the Cortex-M4. In addition to that, Cortex-M4 target device has the most successful faults among the injections compared to the other devices. This could be explained with the fact that this device has more features and microarchitectural elements. Faulty behaviors may appear with different percentages as well. However, we have found that the occurrence probability of a specific behavior can be increased or decreased by fine adjustments of the glitch parameters. Tuning the glitch parameters, for the different devices in order to target different locations of the program could make some faulty behaviors appear or disappear.

Target program dependency

With respect to the target programs, small changes in the target code have large consequences on the observed faults, as noticed for *Program 1* experiments: some faulty

behaviors disappeared, such as `R1 and R3 reset`. New faults appeared, such as `R8 corruption`. Also, different corrupted values are observed: for instance, `R0` had `right shift by 8, 16 or 24 bits` in the first experiment, but it only had `reset` in the second round. In addition to that, in our published work [103], a similar target program was used for the same target device but with different registers, and some of the obtained faults were different. For example, the propagation effect on the `ADD` instruction at line three in Listing 2.1 was obvious, but this time, it is not observed for the Cortex-M4 device. In addition, targeting the same sequence of instructions (*i.e.* the same target part) with two different prologues, even with a simple modification like adding a single `NOP`, could lead to various faulty behaviors, as observed in *Program 2* experiments. As an example, for Cortex-M3 device, `single skip with R6 corruption` occurred only in the first experiment, while `complete instruction skip and repeat` is only observed in the second experiment.

On the difficulty of analyzing the program flow faults

For the second experiment of *Program 1*, one might think that duplicating `CMP` will work as a countermeasure for `APSR corruption` since instruction duplication could work as a software countermeasure as described in [72], [104]. It did not, however, as the injection affects two instructions in most cases, which might be related to the microarchitectural possibility to fetch two instructions at the same time. Hence, the corruption of `APSR` might still occur as a result of either corruption in the second `CMP` or corruption in the first and skipping the second. However, we cannot ensure that a single skip in one of the `CMP` instructions has occurred as executing one of them, either properly or improperly, will mask the single skip effect. Thus, at this step, we can only say that either `double skip` or `APSR corruption` have occurred.

The corruption of `APSR` flags can be due to several causes: a change in the values of the registers while executing `CMP`, an error while decoding the register numbers, an error that occurred when updating the `APSR` flags, a fault in the `ALU` while executing the subtraction between the registers, or a fault in a control signal related to the `APSR` flags. All these hypotheses cannot be validated or discarded without a better knowledge of the microarchitecture or looking at other levels of abstraction, which will help in having a suited fault model at the end.

For all injection campaigns on the two target programs, different forms of `instruction skip` are obtained: `single, double and quad`. This can refer to the possibility to fetch

two or more instructions at the same time or having a prefetch unit that could have a maximum size of 128 bits. More investigation and experiments are needed to uncover the origin of such faults at lower levels of abstraction.

Registers corruption

In terms of the injection effects on the registers, some registers that are not used in the program end up being corrupted as well: `R0`, `R1`, `R3` and `R8` for *Program 1*; `R0` and `R2` for *Program 2*. A question arises about what would be the proper fault model to account for this effect. In particular, such errors may have several causes: it might be related to the instruction opcode (*i.e.*, a fault during the instruction fetch) or to the execution stage of the pipeline. And most importantly, there is no explanation at this level for some corrupted values found in the registers, either used or not in the target part of the programs: `0, values related to the program counter, shifted or seemingly random`. We believe that some of these values are related to the microarchitecture, which will affect how a corrupted instruction will be executed.

Some observed faulty values, however, can be explained as a `source operand replacement`. For example, in *Program 2* experiments, some corrupted values in `R4` were the result of the sum of `R0` and `0xB` or the sum of `R3` and `0xB` instead of `R4` and `0xB`. The former case can occur due to a fault in the decode stage (`R0` instead of `R4`); while the latter can occur due to a fault of not updating one of the inputs to the arithmetic logic unit in the execute stage (`R3` was just used in the previous instruction as shown in Listing 2.2). This explanation cannot be confirmed without further investigations.

State-of-the-art fault models reproducibility

A very interesting point is also observed: using clock glitch fault injection, we were able to observe faulty behaviors that were obtained in the literature using other fault injection techniques. For example, `skip, repeat, and source operand substitution` were observed in [8] using EM fault injection, although, in their experiments, they used superscalar microarchitecture: Cortex-A9. Also, `skip and replay` faults were observed in [92] as a result of performing laser fault injection on a microcontroller that embeds Cortex-M0+. Such a result could help researchers to study the effects of costly fault injections using low-cost equipment and techniques such as clock glitch.

Summary

Finally, the aforementioned faults could be exploited as vulnerabilities in a security application. For example, an `APSR corruption` can lead to test-inversion where tests are considered very important in the control-flow of critical applications.

To sum up, we saw how fault characterization is difficult based on a single level of analysis. These results show the difficulty of building consistent fault models that allow designers to predict the fault injection effects and design efficient and cost-effective countermeasures. Thus, additional research is necessary. In the next section, we propose a methodology that takes into consideration multiple levels of analysis by including software and RTL fault simulations as well as physical fault injections. This will help in explaining the observed points and answering the above-mentioned questions.

2.2 Proposed methodology

This section provides a full description of the proposed methodology to validate and infer fault models that will help in designing hardware and software countermeasures at an optimal cost. In addition to that, these fault models can be used to evaluate vulnerabilities of software codes and/or hardware designs with respect to fault injection attacks. This methodology deals with three different levels of understanding in order to provide a cross-layer fault analysis.

Fig. 2.6 depicts the proposed methodology. It is centered around a comparison between the obtained results from performing three distinct operations of fault injections and simulations: physical fault injection, RTL fault simulation and software fault simulation, in order to make decisions about the consistency and applicability of RTL and software fault models. In other words, starting from the observations obtained at the lowest level of abstraction (*i.e.*, from physical fault injection), it will be possible to optimize fault models at the RTL level, for example, by removing RTL models that do not correspond to experimental observable faulty outputs. Similarly, the models at software level will be optimized, by adjusting them to not include behaviors that cannot be observed at RTL or physical level. This will help in not over-engineering the countermeasures. Also, if a faulty behavior observed from the physical injection does not belong to any faulty output from the RTL simulation, a new RTL fault model must be proposed, or an already existing model must be enhanced. Comparably, a new

software fault model must be inferred, or an existing one must be improved. Thus, this will help in not under-engineering the countermeasures. The following three subsections explain each of the three parts in more details, while the last subsection provides additional discussion on how the proposed methodology is implemented.

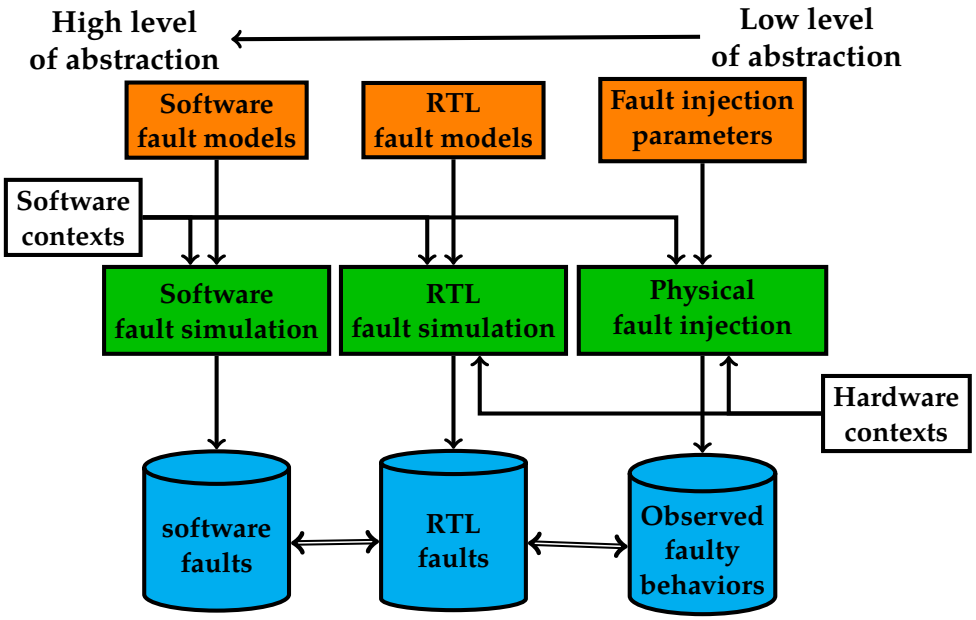


Figure 2.6: Proposed methodology.

2.2.1 Physical fault injection

At this stage, the goal is to perform physical fault injections using a variety of injection techniques using dedicated printed circuit boards and suitable generators. In each injection campaign, the following procedure is applied:

- Define software contexts as target programs for the injection process. Faults are going to be injected while executing these programs on one of the hardware physical targets. These hardware targets include, for instance, microcontrollers, ASICs and FPGAs. A target part (or parts) within each software has also to be defined where faults should be injected.

- Define the set of injection parameters. For example, in the case of clock glitch attacks, the range of values for the shift and the width of the glitch, as well as the delay, as described and explained in the previous section. These parameters as

well as the target device layout must be taken into account when describing the fault model.

- Get a snapshot of the state of the target: for example, the registers and memory states will be put in a known state at the beginning, and these states will be read at the end of the program execution (using a serial communication link with the host computer for example). The richer the information that can be accessed, the more precise the model will be: for instance, hidden performance counters could be used to get a more detailed view of the internal state, in particular when advanced microarchitectural features are implemented. Then, the snapshot will be compared with the configuration of a golden run. The faulty behaviors will be stored in a dedicated file or database, as visually shown in Figure 2.6. This step will allow us to observe the relation between the observed faulty behaviors and the instructions in the target part. In other words, the aim is to assess if there is a direct relation (*i.e.*, the effect corresponds to the target instructions), an indirect relation (*i.e.*, the effect is a result of a propagation effect), or no relation at all, which may require further analysis.

2.2.2 RTL fault simulation

In order to understand what is exactly happening internally at the microarchitectural level and be able to know the origin of a fault, fault simulation campaigns are performed at the RTL description of the target processor. This helps in characterizing further the obtained faulty behaviors by giving more observability and controllability.

With RTL fault simulation, it is possible to inject faults in a very precise manner into the microarchitecture. For instance, inter-stage pipeline registers, multiplexers and different arithmetic units that are involved in executing an instruction in the pipeline stages can be targeted. The fault simulation will consist in forcing the corresponding signals according to existing fault models such as single or multiple bit-flips, bit-sets and bit-resets. Nonetheless, and as mentioned earlier, these fault models will be assessed to better describe the fault effect. To put it in another way, if a new fault model can be inferred to better describe and understand the faulty behavior at RTL, then this new model will be the one to be used and proposed. Therefore, a new fault model could replace an existing or classical fault model that could be either an unexplained model or a very generic model.

As in the case of physical injection, the resulting RTL faults will be stored in a dedicated file or database and then be compared with those observed from the physical fault injections. To ease the comparison and the fault characterization at the RTL level, a divide-and-conquer approach is used to reduce the complexity: the fault simulation is applied to a specific RTL module or specific microarchitectural component at once. Additionally, to better assess and understand the fault effect at hardware level in general and at RTL in particular, other kinds of hardware fault simulation are conducted, for example, post-syntheses timing simulation. This would necessarily help in assessing or improving an existing fault model, or even proposing a new model that better describes the fault effect. More details are provided in chapter 4.

The comparison process helps in two aspects, as shown visually in Figure 2.7. On the one hand, this aims at explaining at the hardware level the faulty behaviors observed from physical injections, and hence, making the fault effect characterization easier. The explanation is done by revealing the origin of the fault at the RTL level and determining the responsible microarchitectural component, the register, or even the single flip-flop behind obtaining the faulty behavior resulting from injecting the fault. On the other hand, it also helps in validating and assessing the realism of the used RTL fault models. Hence, it provides a full overview to the hardware designer in order to build the required countermeasures.

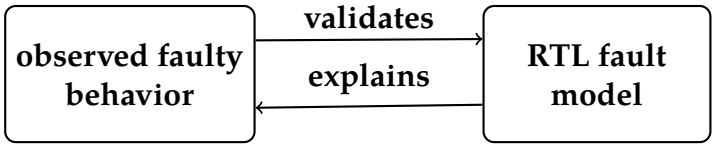


Figure 2.7: Relation between observed faulty behavior and RTL fault model.

2.2.3 Software fault simulation

Software faults will be injected into different target programs. This can be done by performing modification, deletion or addition of instructions in the original program. The software faults may correspond to a large variety of faulty behaviors modeled at the ISA level. This includes, for example, simulating the fault at the instruction level, or even on the binary encoding of the instruction. It also provides description for the fault effect at higher level, when software applications are targeted for exploitation or evaluation purposes.

Firstly, typical fault models such as instruction skip, instruction replacement, instruction corruption, register value corruption, test-inversion, or a combination between these models, are injected into the programs by modifying the instructions. However, inference of new fault models, or improvement of existing fault models issued from other levels, will be applied to better understand the fault effects at software level, and also to cover any observed fault behavior where no corresponding fault model existed beforehand.

The expected faulty outputs will again be stored in a corresponding database. Then, a comparison process similar to the one mentioned earlier will take place between the RTL and the software faulty results on the one hand, and on the other hand, a comparison with the obtained faulty behaviors from physical injection will take also place. Thus, an RTL model (and/or an observed faulty behavior) validates the consistency of a software model, whereas a software model will be usable to describe the occurrence and explain an RTL model (and/or an observed faulty behavior) at application level, which makes the fault effect characterization at this level easier.

2.2.4 Discussion

It must be noted that the comparison between the results of injections and simulations is not only one-way direction, or only between two levels; the comparison takes into account all levels as shown in Figure 2.8. This also includes analyzing the effect on sub-levels. For instance, for software level, this includes binary encoding of instructions, assembly, and application levels. Additionally, there is no specific order that must be followed while comparing and analyzing the results. This is because the implementation of the methodology will follow an iterative flow. In other words, whenever a fault model is improved or a new fault model is inferred at a specific level, simulations and injections on other levels must be conducted to validate this enhanced or proposed fault model.

For the sake of proposing new realistic fault models, fault model inference approach is applied, as shown in Figure 2.9. This approach is part of the whole methodology, which particularly focuses on inferring fault models while making sure of their realism. Therefore, they are able to produce results similar to the results observed from physical injection. More precisely, physical fault injections are performed, with appropriate injection parameters, on a target device that is executing a target program (step ① in

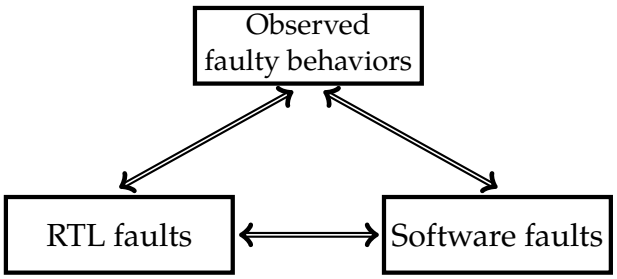


Figure 2.8: Comparison of simulations and injection results

Figure 2.9). Then, from the physical fault injection results, we infer fault models at a specific abstraction level (step ② in Figure 2.9). Applying the inferred fault models to perform fault simulation over the same target program, that was used in step ①, is the next step (step ③ Figure 2.9). The outcomes of the physical fault injection and the fault simulation are then compared, as before, in order to validate the inferred fault models (step ④ in Figure 2.9).

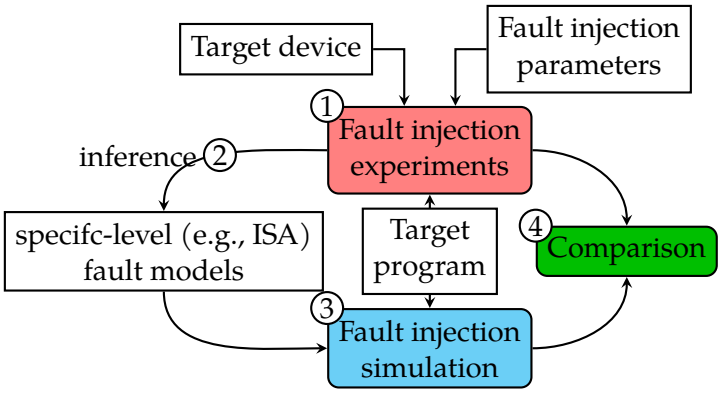


Figure 2.9: Fault model inference approach.

Finally, it is important to highlight that the proposed methodology is independent in terms of physical fault injection technique, target device, and target program. In our scenarios, over this dissertation, we chose embedded microcontrollers that are widely used in the IoT domain, and represent thus a significant test case. It is also worth mentioning that authorized access to the architectures used in this work is provided under the Arm Academic Access (AAA) agreement. This allows us to apply the RTL fault simulation to the three architectures: Cortex-M0, Cortex-M3 and Cortex-M4. Regarding the injection technique, chapters 3 and 4 provide results of clock glitch fault injection, while chapter 5 presents results of voltage glitch fault injection.

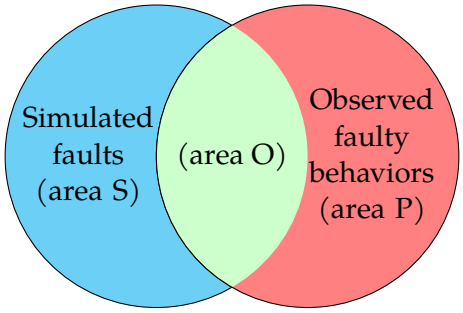


Figure 2.10: Faults generated from simulation and injection with overlapping area.

2.3 Fault models evaluation

For the sake of assessing and validating the proposed fault models, Laurent *et al.* [95], [105] defined two metrics: Coverage and Fidelity. In this work, we also use these two metrics. In addition to these two quantitative metrics, we define a new metric called Complexity to introduce a qualitative assessment of the complexity of a fault model. These metrics evaluate a proposed fault model at a specific level of abstraction with respect to the observed faulty behaviors resulting from physical fault injection. In other words, the observed faulty behaviors represent the reference to measure these metrics. Conversely, in the work of Laurent *et al.*, they measured Coverage and Fidelity based only on simulations, where the results of RTL fault simulation were the reference to assess software fault models. The following subsections describe these metrics in detail.

2.3.1 Coverage

It measures the portion of the observed faulty behaviors that the used fault models accurately predicted. Thus, it represents the faulty behaviors that fall in the overlapping area in Figure 2.10. The circle on the right in Figure 2.10 corresponds to the observed faulty behaviors from physical fault injection, while the circle on the left represents the faults generated from fault simulation while applying a set of fault models at a specific level of abstraction.

Mathematically, the fault Coverage (C) is defined in Equation 2.1. Where:

- |area P| is the number of observed faulty behaviors from **P**hysical fault injection that are **not** predicted by fault models, *i.e.,* the red area in Figure 2.10.

- |area O| is the number of observed faulty behaviors from physical fault injection that are predicted by fault models, *i.e.,* the **O**verlapping green area in Figure 2.10. |area O| also represents the number of obtained faults from fault simulation that correspond to actual observed faulty behaviors.

$$C = \frac{|area\ O|}{|area\ O|\ +\ |area\ P|} \times 100\% \tag{2.1}$$

2.3.2 Fidelity

It measures the portion of the obtained faults from fault simulation that correctly predicts observed faulty behaviors. Thus, it represents the simulated faults that fall in the overlapping area in Figure 2.10. To better differentiate Fidelity from Coverage with respect to the overlapping area, one should notice that the size of the two circles in Figure 2.10 does not necessarily be the same.

Mathematically, the Fidelity metric (Fd) is defined in Equation 2.2. Where:

- |area O| as defined in subsection 2.3.1.

- |area S| is the number of the obtained faults from fault **S**imulation that does **not** correspond to actual observed faulty behaviors, *i.e.,* the blue area in Figure 2.10.

$$Fd = \frac{|area\ O|}{|area\ O|\ +\ |area\ S|} \times 100\% \tag{2.2}$$

2.3.3 Complexity

This metric provides a qualitative measurement for a fault model. More precisely, it measures how applicable is the fault model in a vulnerability analysis process. Moreover, it takes into account how easy is the fault model to be used by a software developer or hardware designer to develop or design a countermeasure. Also, other properties of a fault model can be considered when assessing its complexity, this includes:

- Description: how generic is a fault model description, is another attribute that can affect the Complexity. For example, "Corruption" fault model is an extremely generic term that surely increases the complexity of a fault model.

- Explainability: providing an explanation of a fault model, at single or multi-abstraction levels, would definitely decrease its complexity. On the other hand, having an unexplained fault model, for instance: "Random corruption" would clearly increase its complexity.
- Target dependency: the independence of a fault model with respect to a target program or a target device would certainly make it a better model, and thus, a less complex model that is applicable to different scenarios, with no additional constraints.

To evaluate the quality of a fault model with respect to the Complexity metric, we will use adjectives that convey the degree or intensity of its difficulty when applying it to evaluate vulnerabilities or to design countermeasures. This includes, for example, easy, hard, very hard, *etc*. Additionally, to assess Complexity with respect to the above-mentioned properties, we may use words, such as, low, high, very low, *etc*.

2.3.4 Summary

In summary, these metrics will absolutely help in evaluating fault models, and hence, refining the inferred fault models. Therefore, realistic fault models will be proposed. Based on the Coverage and Fidelity metrics, the main objective will be to increase the overlapping area in Figure 2.10 as much as possible. However, this might not be enough for the quality of a fault model. Thus, we defined Complexity as an additional metric. The objective for Complexity will be to have the simplest possible fault model that can be used in vulnerability analysis or in countermeasure design. In addition, we seek to obtain the lowest level of Complexity in terms of the aforementioned properties.

2.4 Conclusion

In this chapter, we presented the existing problems in analyzing and understanding fault attacks in complex microarchitectures, while focusing only on ISA level for the analysis. We highlighted this by providing experimental evidence of intrinsically microarchitectural faults, using clock glitch as the fault injection technique. The experimental results showed that the observed faulty behaviors can be target-dependent, prologue-dependent, and architecture-dependent.

After that, we proposed a new methodology to provide a cross-layer analysis for characterizing faulty behaviors. Such methodology can be used to build realistic fault models at different levels, such as RTL and software levels. It can also provide an explanation for the origin of the observed faults. Hence, this gives the possibility to design suited countermeasures at the most appropriate cost at hardware and software levels. Additionally, it makes the process of vulnerability analysis easier.

Finally, metrics to evaluate and assess the realism and the quality of the proposed fault models are provided.

3

Preliminary RTL simulation and new binary encoding fault models

With the increasing complexity of embedded systems, the use of variable-length instruction sets has become essential to achieve higher code density and better performance. However, security aspects are closely linked to this, as attack techniques and equipment continue to improve. One such rising physical attack technique is fault injection. Yet, as detailed in the previous chapters, hardware designers and software developers lack accurate fault models to evaluate the vulnerabilities of their designs or codes in the presence of such attacks.

To follow the proposed methodology, this chapter presents preliminary RTL fault simulation experiments. These experiments demonstrate that the alignment of instructions in memory greatly affects the observed faulty behaviors, due to the target processors supporting variable-length instruction set. As a result, fault models at the binary encoding level of instructions are inferred. These models provide an explanation for a wide range of faulty behaviors that are experimentally observed when a processor running a variable-length instruction set is targeted. The analysis and characterization process include the binary encoding of instructions and show how the obtained behaviors depend on the alignment in memory. Moreover, applying the proposed fault models leads to provide a proof of concept exploitation example, where we were able to break the control flow integrity of a program by modifying the value of the program counter. Additionally, vulnerability analysis on three different implementations of AES encryption algorithm is provided, which serves as a real-life application example.

In the following sections, section 3.1 presents a series of fault simulation experi-

ments, at hardware level, that unravels the relation between the observed faulty behaviors and the alignment of instructions in memory. Section 3.2 provides the necessary background on variable-length instruction sets. Fault models along with experimental setup and findings are detailed in section 3.3. Exploitation and vulnerability analysis scenarios are carried out in section 3.4. A tool to automate the simulation of the fault models and analyze the injection results is described in section 3.5. Evaluation of the proposed fault models is provided in section 3.6. The chapter is concluded in section 3.7.

3.1 Preliminary RTL fault simulation and analysis

To start validating the proposed approach, a preliminary analysis study has been conducted by performing RTL fault simulation experiments on an RTL description version based on the Arm Cortex-M3 processor. This version is called Arm Cortex-M3 DesignStart Eval RTL [106]. It is an open-source version. Its RTL description is for an FPGA and is based on the Arm Cortex-M3 processor. It provides a flexible way for designers to evaluate their embedded system designs using the Arm Cortex-M3 processor. It also enables easy SoC design and simulation, followed by hardware prototyping.

The DesignStart RTL description consists of many subsystems, as shown in Figure 3.1. This includes: preconfigured obfuscated Cortex-M3 processor, memory subsystem, peripherals for application use, *etc*.

Subsection 3.1.1 presents the initial simulation experiments where clock glitch simulation has been conducted on this RTL description, at the behavioral level. As a result, subsection 3.1.2 tries to model the observed faulty behaviors using typical RTL fault models.

It should be mentioned that the RTL simulation experiments in this section have been done before the academic access agreement with Arm (AAA) was signed. Moreover, DesignStart version can be described as a gray-box setting, where it has an obfuscated RTL for the core itself, while non-obfuscated RTL for the surrounded subsystems. Therefore, these simulation experiments are considered preliminary. This is because the obtained results need further analysis, formalization and confirmation from additional experiments on a non-obfuscated RTL description, as will be detailed in chapter 4.

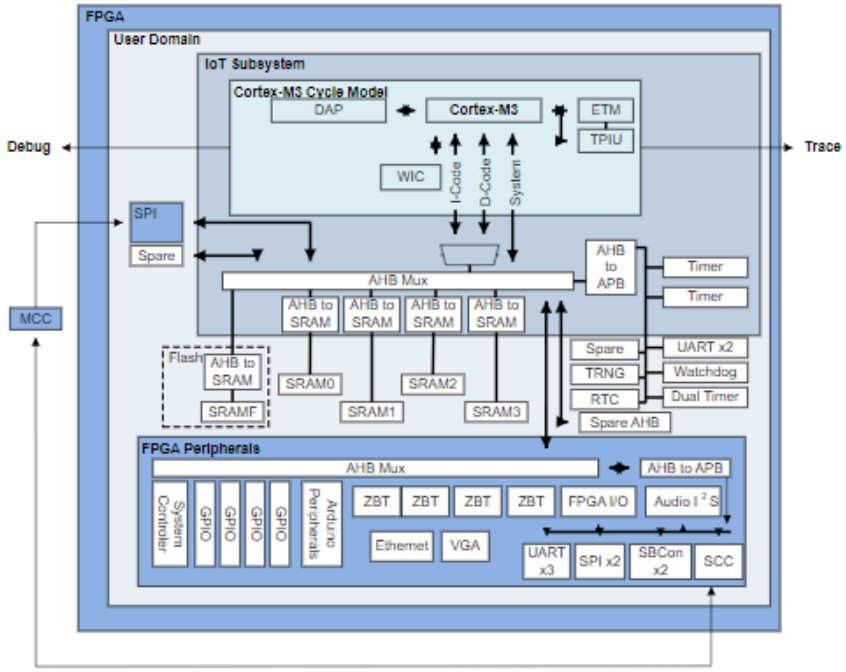


Figure 3.1: Cortex-M3 DesignStart Eval system diagram (from [107])

3.1.1 Internal clock glitch simulation

Since an RTL description may contain thousands of flip-flops, a simple and fast approach is needed to accelerate the fault simulation process. Moreover, as the already observed faulty behaviors are resulting from clock glitch fault injection, a method to simulate the clock glitch is the first thing thought of. In clock glitch fault injection, as previously mentioned, the glitch is injected on the global clock, thus, there is no particular information about which microarchitectural element could be affected as a result of the injection. Fortunately, as DesignStart version consists of multi-subsystems, it is possible to simulate the glitch for a specific clock that is fed to a specific subsystem without affecting other clocks. This allows simulating the glitch on an internal clock signal. Hence, creating a timing violation between a subsystem and other subsystems within the whole system. This is because the glitch manifested as an additional clock cycle exclusively for the target subsystem, while other subsystems are under the normal clock cycle.

The simulation is performed by forcing an internal clock signal to have `Logic 1` for a given period of time in a specific clock cycle, where the clock signal is originally having

a `Logic 0`, between a falling edge and the following rising edge, as shown in Figure 3.2.

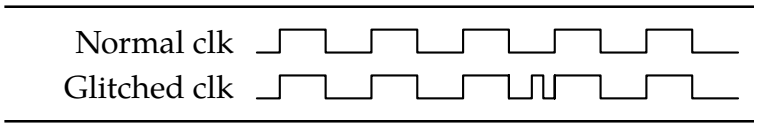


Figure 3.2: Clock glitch simulation.

Simulating the clock glitch, while executing the data-flow target program used in chapter 2 (also shown in Listing 3.1), allows obtaining faulty behaviors similar to some of the observed faulty behaviors resulting from physical clock glitch fault injection. This is achieved when the faulty clocks are either: the system clock that is fed to the AHB (Advanced High-Performance bus) and the processor subsystem (IoT Subsystem in Figure 3.1). This clock is called `CPU0HCLK`, also faulting its successor (`HCLK`) led to the same kind of faulty behaviors. Or, the clock that is fed to the Flash memory (Flash Subsystem surrounded by a dashed rectangle in Figure 3.1). This clock is called `SRMAFHCLK`. Furthermore, adding a single `NOP` at the beginning of Listing 3.1, and performing the same clock glitch simulation again, allows obtaining different faulty behaviors that are also similar to observed behaviors from physical clock glitch fault injection.

```
1 ADD R1, R1, 0x6  // r1 = r1 + 0x6
2 ADD R3, R3, 0xa  // r3 = r3 + 0xa
3 ADD R4, R4, 0xb  // r4 = r4 + 0xb
4 ADD R5, R6, R3   // r5 = r6 + r3
5 ADD R3, R3, 0xf  // r3 = r3 + 0xf
```

Listing 3.1: Target part used in the internal clock glitch simulation.

Based on these observations, the goal is now to find the RTL registers or signals that allow obtaining these faulty behaviors, and hence, being able to model these faults at RTL, as will be described in the next subsection.

3.1.2 RTL fault simulation using bit manipulation fault models

While analyzing how the clock glitch simulation affected RTL signals, it is found that some signals are erroneously updated at the rising edge of the glitch. These signals

are located in both directions of the path between the Flash memory and the core. Figure 3.3 shows this path in a schematic way. This path is the instructions' data path, where the data propagate while fetching the instructions.

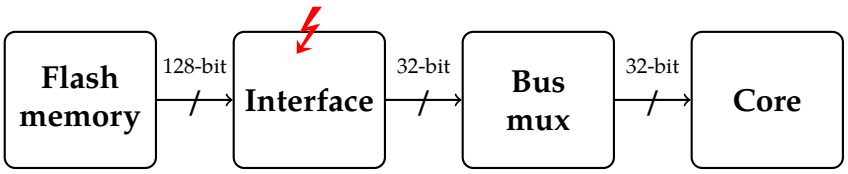


Figure 3.3: Instructions' fetch data path from the Flash memory to the processor core.

Using classical RTL fault models, we were able to observe various faulty behaviors similar to real. For example, by single or double bit-set or bit-reset to specific signals that are located in the fetch path, in particular in the interface, as shown in Figure 3.3, we were able to observe the following faults:

- Skip an instruction and repeat the previous one.
- 128-bit complete instructions skip and repeat the previous 128-bit block. This behavior is described in [74].

Adding a single `NOP` at the beginning of the target part in Listing 3.1 and applying the same RTL fault model, that led previously to *Skip an instruction and repeat the previous one* fault, allowed observing a totally different faulty behavior, which can be described as *double instruction corruption*. This outcome was also similar to a realistic behavior observed from physical clock glitch fault injection. As a result of analyzing the related faulty signals, the rationale behind such observation was figured out: the instructions' data (*i.e.,* the binary encoding of the instructions) that are carried by these signals (*e.g.,* `HRDATAI`) did not necessarily belong to complete instructions. For instance, they might correspond to two different instructions. This is because the supported ISA offered *variable-length* instructions. Thus, misaligned instructions can be fetched. This knowledge led us to infer various fault models, at the *binary encoding level* of the instructions. These models allowed explaining a wide range of the observed faulty behaviors from physical injections. The models and the related experimental results are detailed within this chapter in the following sections. Before digging into details, the next section provides a background on variable-length instruction sets.

3.2 Variable-length instruction sets

Reducing code size is a well-known method to reduce power consumption and memory usage. It lowers the overall cost of an embedded system, highly affected by program coding and the fetch stage in the pipeline [108]. Code size reduction, targeting the highest possible code density, can be achieved by using a variable-length instruction set [108]–[111]. Additionally, less power is consumed due to the smaller number of fetches [109].

A variable-length instruction set can be defined as a combination of two instruction sets:

- a first set of short instructions (with respect to their encoding), mainly used for common operations, while providing the same functionality as if possessing larger encoding;
- a second set composed of instructions with a larger encoding that cannot be compacted while giving the same functionality.

For this reason, the first set can also be referred to as the *compressed set*. High code density is achieved by the compressed instructions, while the second set allows for the preservation of high performance and expressive power. An example of the effect of a variable-length ISA on cost and performance is the instruction cache: shorter encoding needs smaller caches for the same performance [111], [112]. Therefore, having a shorter encoding induces fewer cache misses, with a given cache size, thus increasing the overall throughput of the processor. On the other hand, dealing with different lengths of encoding increases the complexity of the instruction decoder [112].

Several variable-length instruction sets exist. The x86 [113] instruction set, supported by Intel and AMD processors, offers various lengths of encoding from 1 to 15 bytes. Another example of a variable-length instruction set is microMIPS [110], providing a set of 16-bit instructions that correspond to the most commonly used ones, in addition to all the instructions from the MIPS32/64 instruction sets [114], [115]. MicroMIPS shows 35 % smaller code size, although almost the same performance as MIPS32 [110]. In 2015, a draft proposal [111] was published to procure 16-bit encodings for some instructions in the RISC-V instruction set. This new instruction set has been known as RISC-V Compressed (RVC) and reduces the code size by more than 25 % [111]. RVC extension is now part of the recent RISC-V specifications [116].

Finally, most Arm processors, including Cortex-M3, Cortex-M4 and Cortex-A9, support a dedicated variable-length instruction set as well, known as Thumb2 instruction set [102]. It consists of two sets of 16-bit and 32-bit instructions. Thumb2 delivers 30 % of code size reduction on average [109].

In this work, Cortex-M3 and Cortex-M4 have been chosen as target processors, for their wide adoption in embedded systems. Thumb2 is the target instruction set, but it is straightforward to generalize our findings to other variable-length instruction sets, as the fetching mechanisms are similar in most devices regardless of the alignment of the code in memories. For example, Intel and AMD architectures usually support fixed-size of caching blocks called *cache lines* [117], [118]. Consequently, as these cache lines have fixed size, Intel and AMD architectures allow splitting the instructions' data over the cache lines. This splitting may occur when an instruction spans across two or more cache lines. In such cases, the processor fetches the necessary cache lines to obtain the complete instruction.

3.3 Inferred binary encoding fault models

In order to investigate the effects of fault injection on a variable-length instruction set, several physical fault injection experiments have been carried out. They aim at confirming the analysis of the preliminary simulation to provide better characterization and description, at ISA level, for a wide range of faulty behaviors obtained when performing fault injection campaigns. The following subsections present the experimental setup, the experimental results and the related discussion and analysis.

3.3.1 Experimental setup

Clock glitch fault injection experiments have been conducted using the ChipWhisperer environment. In the following, the target devices and programs are described.

Target devices

The target devices are two 32-bit microcontrollers: STM32F1, which embeds an Arm Cortex-M3 processor, and STM32L4, which embeds an Arm Cortex-M4 processor.

The Arm Cortex-M3 and Cortex-M4 cores both include a 3-stage pipeline: fetch, decode, and execute. Both are based on the ARMv7-M [77] architecture and support the Thumb2 instruction set, consisting of variable-length instructions as mentioned in the previous section: 16-bit and 32-bit instructions.

In the Arm Cortex-M3 device, the fetch size from the memory to the AHB (Advanced High-performance Bus) is fixed and equal to 32 bits, regardless of the size of the instruction. Hence, as a result of having variable-length instructions, the fetched 32 bits can belong to one of the cases in Figure 3.4 or Figure 3.5. Figure 3.4 represents the fetching cases when code is aligned in memory, while Figure 3.5 represents the misaligned cases.

The Arm Cortex-M4 device supports cache lines of 64 bits. Therefore, in this case, the flash memory access size is 64-bit wide. Consequently, the fetched 64 bits from the memory to the cache line can belong to two combined blocks of cases in Figure 3.4 or Figure 3.5. How these different possibilities affect the observed execution, as a reaction of fault injection campaigns, will be addressed in Section 3.3.2.

The processor detects whether the instruction that is about to be executed is a 16-bit or 32-bit one by analyzing the five most significant bits of the half-word that arrives first [77]. If these five bits have one of the following three values, then the word contains a 32-bit instruction:

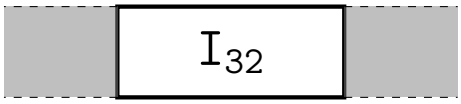
- `0b11101`,
- `0b11110`,
- `0b11111`.

All the other values define a 16-bit instruction. This understanding is central to unravel the monitored faulty behaviors, as detailed in Section 3.3.2.

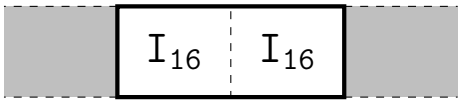
In the following sections and chapters, the big-endian representation for the binary encoding of instructions is used for readability. Thus, for a 32-bit instruction, the most significant 16 bits arrive *first* in the pipeline.

Target programs

As in chapter 2, the injection is performed into inline assembly instructions within a C program. Also, the program is divided into three parts, referred to as Prologue, Target, and Epilogue, and separated by `NOP` instructions to ease the fault injection process.



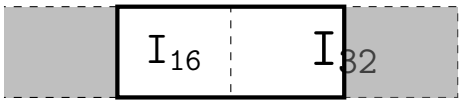
(a) Fetching one 32-bit instruction.



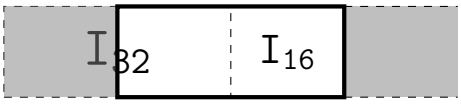
(b) Fetching two 16-bit instructions.

Figure 3.4: Fetching aligned instructions.

(a) Fetching the bottom half of a 32-bit instruction and the top half of another 32-bit instruction.



(b) Fetching one 16-bit instruction and the top half of a 32-bit instruction.



(c) Fetching the bottom half of a 32-bit instruction and one 16-bit instruction.

Figure 3.5: Fetching misaligned instructions.

In the experiments, arithmetic instructions have been used in the Target part as shown in Listings 3.2 and 3.3. These instructions are similar to the target instructions in Listing 3.1, however, two instructions are added at the beginning of the Target part, *i.e.* `MOV` and `LSLS`. These two instructions are 16-bit instructions. Conversely, all the `ADD` instructions are 32-bit instructions. The additional two instructions aim at clarifying the effect of the injection on variable-length instructions. Moreover, they also illustrate how the code alignment in memory greatly affects the resulting faulty behaviors.

Listing 3.2 shows an example of an *aligned* code. In this case, when 32 bits are fetched, they are either two full 16-bit instructions or one 32-bit instruction, as shown in Figure 3.4. At the end of a normal execution, each register has a different value from the others. Therefore, faults can be more easily identified whenever they occur. Various small immediate values are used with the `ADD` instructions to assess whether one would be replaced by a register number or vice versa, for example, whether `R3` becomes `0x3`. More details are given in 3.3.2.

```
1 MOV  R8, R4         // R8 = R4
2 LSLS R2, R0, 0x10   // R2 = R0 « 0x10
3 ADD  R1, R1, 0x6    // R1 = R1 + 0x6
4 ADD  R3, R3, 0xa    // R3 = R3 + 0xa
5 ADD  R4, R4, 0xb    // R4 = R4 + 0xb
6 ADD  R5, R6, R3     // R5 = R6 + R3
7 ADD  R3, R3, 0xf    // R3 = R3 + 0xf
```

Listing 3.2: Target part in the aligned code.

A *misaligned* code is illustrated in Listing 3.3. It is similar to Listing 3.2 except that the first `MOV` instruction has been removed. Only one 16-bit instruction is now fetched. Therefore, the code is misaligned. When 32 bits are fetched, they now belong to either two different 32-bit instructions, or one 16-bit instruction and half of a 32-bit instruction, as shown in Figure 3.5. Section 3.3.2 will develop how such a small modification of the Target code can greatly affect the observed faulty behaviors at ISA level.

```
1 LSLS R2, R0, 0x10   // R2 = R0 « 0x10
2 ADD  R1, R1, 0x6    // R1 = R1 + 0x6
3 ADD  R3, R3, 0xa    // R3 = R3 + 0xa
4 ADD  R4, R4, 0xb    // R4 = R4 + 0xb
5 ADD  R5, R6, R3     // R5 = R6 + R3
6 ADD  R3, R3, 0xf    // R3 = R3 + 0xf
```

Listing 3.3: Target part in the misaligned code.

The Prologue is always aligned in the code memory space and does not influence the overall code alignment, which only depends on the instructions of the Target part.

Injection parameters

Each injection campaign involves repeating the clock glitch fault injection 10 000 times for each combination of glitch parameters to maximize the number of captured faults. Thus, for each of the presented examples in Section 3.3.2, 10 000 executions are conducted. Table 3.1 shows the shift and width values that allowed observing the presented faulty behaviors, for each target device. The values are expressed in percentage of one clock period. The negative value of the shift means that the glitch is injected before the rising edge of the target clock cycle. These shift and width values reproduce

similar faulty behaviors when targeting other microcontrollers of the same series (*i.e.* other STM32F1 and STM32L4). The glitch is timed accordingly to target specific instructions. Thus, the delay value depends on the target instruction(s), the number of instructions in the Prologue, and the number of `NOP` instructions that precede the Target part. A *single* glitch is injected during each program execution.

	STM32F1			STM32L4		
faulty behavior	shift	width	times	shift	width	times
skip 32 bits aligned	-49	4	9.06 %	-14	10	92.68 %
skip 64 bits aligned	-	-	-	-13	10	100 %
skip & repeat 32 bits aligned	-13	3	99.8 %	-	-	-
skip & repeat 64 bits aligned	-	-	-	-6	2	75.98 %
skip 32 bits misaligned example1	-49	5	1.97 %	-	-	-
skip 32 bits misaligned example2	-12	3	99.51 %	-12	6	100 %
skip 64 bits misaligned	-	-	-	-13	10	100 %
skip & repeat 32 bits misaligned	-13	3	99.98 %	-	-	-
skip & repeat 64 bits misaligned	-	-	-	-6	2	65.34 %

Table 3.1: Shift and width values used in Section 3.3.2 experiments, and percentage of occurrence of each faulty behavior over 10 000 executions.

In the next section, we focus on an observable subset of captured faults, discarding crashes and silent cases. This subset is the largest subset of the obtained faults, and our focus is on the occurrence of these faults, not their probability. Nonetheless, Table 3.1 shows the percentage of occurrence for each of the presented faulty behaviors over 10 000 executions.

3.3.2 Experimental results and analysis

Different faulty behaviors have been observed after performing clock glitch fault injection campaigns. In the following three subsections, we only focus on faulty behaviors related to two specific inferred fault models: the ones referring to the *encoding* of the instructions “Skip” or “Skip and repeat” a specific number of bits, which can either be 32 or 64 bits. Other special faulty behaviors, also related to the encoding of the in-

structions, are described in the last subsection of section 3.3.2. These additional faulty behaviors depend on the target device. Thus, extra fault models are provided.

In the following presented results, all the occurrences of 32-bit faulty behaviors were observed when targeting the Arm Cortex-M3 device. Some of them were also monitored when targeting the Arm Cortex-M4 device, as shown in Table 3.1. On the other hand, 64-bit faulty behaviors were only obtained when targeting the Arm Cortex-M4 device. Their fetch size is the reason, as previously detailed in section 3.3.1.

Aligned code scenario

Listing 3.4 represents the binary encoding of the target program previously shown in Listing 3.2. Each line corresponds to one 32-bit instruction, except line 1, which corresponds to the two 16-bit instructions. Therefore, this code is *aligned* in memory.

```
1 46a00402 // MOV R8, R4 // LSLS R2, R0, 0x10
2 f1010106 // ADD R1, R1, 0x6
3 f103030a // ADD R3, R3, 0xa
4 f104040b // ADD R4, R4, 0xb
5 eb060503 // ADD R5, R6, R3
6 f103030f // ADD R3, R3, 0xf
```

Listing 3.4: Binary encoding for the aligned code in hexadecimal format.

Assuming that i is the line number that points to a 32-bit block of the target program binary encoding , then the “Skip” and “Skip and repeat ” fault models can be detailed as such:

- Skip 32 bits: the 32 bits at line i are skipped, and the execution resumes from line $i + 1$.
- Skip & repeat 32 bits: the 32 bits at line $i + 1$ are skipped and the 32 bits at line i are repeated.
- Skip 64 bits: the 32 bits at line i and the 32 bits at line $i + 1$ are skipped, and the execution resumes from line $i + 2$.
- Skip & repeat 64 bits: the 32 bits at line $i + 1$ and the 32 bits at $i + 2$ are skipped, while the 32 bits at line i, and the 32 bits at line $i - 1$ are repeated.

The following subsections show samples of the observed "Skip" and "Skip and repeat" faulty behaviors after performing the fault injection campaigns.

Although we target a given instruction inside the target part for discussion, the conclusions drawn from these examples can also be applied to other locations of the target program without any loss of generality. For each subsection, faults are described at two different abstraction levels: binary encoding and ISA levels. Note that *all* the presented examples have been experimentally observed during clock glitch fault injection campaigns.

Skip 32 bits / single instruction skip Except line 1 related to the binary encoding of two 16-bit instructions, skipping any line in Listing 3.4 has led to a single instruction skip. This is because each line corresponds to a full 32-bit instruction. Skipping the `ADD R4, R4, 0xb` instruction at line 4 in Listing 3.4 is an example; the observed execution at ISA level is reported in Listing 3.5.

```
1 MOV R8, R4
2 LSLS R2, R0, 0x10
3 ADD R1, R1, 0x6
4 ADD R3, R3, 0xa
5 ADD R5, R6, R3
6 ADD R3, R3, 0xf
```

Listing 3.5: Observed execution for Skip 32 bits / single instruction skip.

Naturally, skipping line 1 in Listing 3.4 has led to a double instruction skip since this line corresponds to two 16-bit instructions.

Skip 64 bits / double instruction skip Skipping line 2 and line 3 in Listing 3.4 has brought a double instruction skip since these two lines contain two 32-bit instructions. Listing 3.6 shows the observed execution at ISA level for this sample.

Skip & repeat 32 bits / single instruction skip & single instruction repeat Skipping the `ADD R3, R3, 0xa` instruction at line 3 in Listing 3.4 and repeating the `ADD R1, R1, 0x6` instruction at line 2 is an illustration of this fault model. The monitored execution at ISA level is exposed in Listing 3.7.

```
1 MOV R8, R4
2 LSLS R2, R0, 0x10
3 ADD R4, R4, 0xb
4 ADD R5, R6, R3
5 ADD R3, R3, 0xf
```

Listing 3.6: Observed execution for Skip 64 bits / double instruction skip.

```
1 MOV R8, R4
2 LSLS R2, R0, 0x10
3 ADD R1, R1, 0x6
4 ADD R1, R1, 0x6
5 ADD R4, R4, 0xb
6 ADD R5, R6, R3
7 ADD R3, R3, 0xf
```

Listing 3.7: Observed execution for Skip & repeat 32 bits / single instruction skip & single instruction repeat.

Skip & repeat 64 bits / double instruction skip and double instruction repeat Skipping line 4 and line 5, and repeating line 2 and line 3 in Listing 3.4 is an example of this fault model. The observed execution at ISA level is shown in Listing 3.8.

```
1 MOV R8, R4
2 LSLS R2, R0, 0x10
3 ADD R1, R1, 0x6
4 ADD R3, R3, 0xa
5 ADD R1, R1, 0x6
6 ADD R3, R3, 0xa
7 ADD R3, R3, 0xf
```

Listing 3.8: Observed execution for Skip & repeat 64 bits / double instruction skip & double instruction repeat.

Misaligned code scenario

The focus is now on *misaligned* code. It is achieved by removing the leading `MOV` instruction, a 16-bit instruction, from the Target part in the program as displayed in Listing 3.3. Listing 3.9 shows the binary encoding of the misaligned code.

Each line still contains 32 bits, but unlike the previous case, it does not correspond to a single 32-bit instruction. For the sake of clarity, each instruction has been highlighted with a different color: the reader will notice that each 32-bit instruction is split over two consecutive lines.

```
1 0402f101
2 0106f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00 // bf00: NOP.
```

Listing 3.9: Binary encoding for the misaligned code in hexadecimal format.

There again, similar fault models are used on 32-bit or 64-bit data. However, as a line now consists of two 16-bit blocks belonging to two separate instructions, as previously shown in Figure 3.5, different faulty behaviors have been examined at ISA level. The actual recorded faulty behaviors depend on the target location of the glitch injection. The following subsections provide various observations of each model, *i.e.* different examples of "Skip" and "Skip and repeat" models. It is interesting to highlight that the monitored behaviors in this scenario are significantly more complex than in the aligned code scenario. Among several outcomes, for instance, we have witnessed double instruction corruption or even new instruction execution.

Skip 32 bits / single instruction skip and single instruction corruption This case refers to Figure 3.5a, and happens when skipping line 3 in Listing 3.9 for example. The observed execution at ISA level is in Listing 3.10.

```
1 LSLS R2, R0, 0x10
2 ADD R1, R1, 0x6
3 ADD R4, R3, 0xb  // f103040b
4 ADD R5, R6, R3
5 ADD R3, R3, 0xf
```

Listing 3.10: Observed execution for Skip 32 bits / single instruction skip and single instruction corruption.

The `ADD R4, R4, 0xb` instruction is skipped and the `ADD R3, R3, 0xa` instruction is corrupted, replacing two of its operands by the corresponding ones from the skipped instruction.

Skip 32 bits / double instruction skip and new instruction execution Figure 3.5b illustrates this case; Line 1 in Listing 3.9 is skipped for example. As a result, `0x0106` arrives first to the core and, since the five most significant bits of `0x0106` are `0b00000`, a 16-bit instruction is executed, with the following encoding: `0x0106`. This instruction is `LSLS R6, R0, 0x4`. The other instructions in the target program are not affected and are executed normally.

Listing 3.11 displays the monitored execution of this example at ISA level. The first instruction is painted blue since its encoding comes from the original blue instruction. It is shown that two instructions have been skipped: a 16-bit and a 32-bit one; a *new* 16-bit instruction has been executed instead.

```
1 LSLS R6, R0, 0x4 // 0106
2 ADD R3, R3, 0xa
3 ADD R4, R4, 0xb
4 ADD R5, R6, R3
5 ADD R3, R3, 0xf
```

Listing 3.11: Observed execution for Skip 32 bits / double instruction skip and new instruction execution.

To prove that the new `LSLS` instruction is not related to the original `LSLS` instruction, section 3.3.2 supplies a variety of instructions that can be executed as a result of this observed behavior.

Skip 64 bits / single instruction corruption, double instruction skip and new instruction execution This case relates to two consecutive blocks of Figure 3.5a, and happens when skipping lines 2 and 3 in Listing 3.9 for instance. Listing 3.12 shows its observed execution at ISA level.

The `ADD R1, R1, 0x6` is corrupted after skipping its bottom half (`0x0106`). Since `0xf101` means a 32-bit instruction is about to be executed, it should be padded with another 16-bit block. However, in this case, it is not padded with 16 bits from an upcoming or skipped instruction, as observed previously. Instead, it is padded with zeros:

```
1 LSLS R2, R0, 0x10
2 ADD R0, R1, 0x0  // f1010000
3 LSLS R3, R1, 0x10 // 040b
4 ADD R5, R6, R3
5 ADD R3, R3, 0xf
```

Listing 3.12: Observed execution for Skip 64 bits / single instruction corruption, double instruction skip and new instruction execution.

0x0000. This might be due to the unavailability of another 16-bit block from the bus or in the fetch unit, when the decision of executing a 32-bit instruction is taken. The ADD R3, R3, 0xa instruction is skipped as all of its bits had been discarded. 0x040b, the remaining part of the original ADD R4, R4, 0xb instruction, is executed as a *new* 16-bit instruction, since the five most significant bits, 0b00000, identify a valid 16-bit instruction: LSLS R3, R1, 0x10.

The skipped 32 bits (0x030af104) might also be replaced by zeros. Here it is worth mentioning that the encoding of 0x0000 belongs to a 16-bit dummy instruction that has no effect: MOVS R0, R0. Thus, executing 0x00000000 would have no effect.

This behavior is also observed when considering 32 bits, not only 64 bits, when the code is misaligned in memory.

Skip & repeat 32 bits / double instruction corruption This section refers to Figure 3.5a as a reference; when two half instructions are affected. By way of illustration, in Listing 3.9 it happens when line 4 is skipped and line 3 is repeated.

```
1 LSLS R2, R0, 0x10
2 ADD R1, R1, 0x6
3 ADD R3, R3, 0xa
4 ADD R3, R4, 0xa  // f104030a
5 ADD R5, R4, 0x3  // f1040503
6 ADD R3, R3, 0xf
```

Listing 3.13: Observed execution for Skip & repeat 32 bits / double instruction corruption.

As a consequence, two instructions are corrupted, as shown in Listing 3.13. 0xf104 means that the instruction to be executed is a 32-bit one, as the five most significant bits are 0b11110. The repeated red section (0x030a) is part of the new executed instruction.

In addition, since `0xf104` is repeated, another *new* 32-bit instruction is executed. Its first half is from the `ADD R4, R4, 0xb` instruction (`0xf104`) and its second one is from the 16 bits that remained from the `ADD R5, R6, R3` instruction at line 5 in Listing 3.9 (`0x0503`).

To report the observed behaviors at ISA level and generalize the obtained faults to other target programs with similar structure of encoding, the corruption of two 32-bit instructions is explained as follows:

- For the `ADD R4, R4, 0xb` instruction: the destination operand and the second source operand are replaced by the corresponding operands from the *previous* instruction.
- For the `ADD R5, R6, R3` instruction: the first source operand is replaced by the first source operand from the *previous* instruction. Its opcode (`ADD with register`) is also replaced by the *previous* opcode (`ADD with immediate`). Therefore, register number `R3` is now considered as an immediate value: `0x3`.

Skip & repeat 64 bits / double instruction corruption, single instruction skip and single instruction repeat This case concerns two consecutive blocks of Figure 3.5a, when two 32-bit blocks are skipped and two 32-bit blocks are repeated; it happens when skipping lines 4 and 5 and repeating lines 2 and 3 in Listing 3.9 for example. Listing 3.14 displays its recorded execution at ISA level.

```
1 LSLS R2, R0, 0x10
2 ADD R1, R1, 0x6
3 ADD R3, R3, 0xa
4 ADD R1, R4, 0x6  // f1040106
5 ADD R3, R3, 0xa
6 ADD R3, R4, 0xf  // f104030f
```

Listing 3.14: Observed execution for Skip & repeat 64 bits / double instruction corruption, single instruction skip and single instruction repeat.

The `ADD R4, R4, 0xb` instruction is corrupted, as the bottom half of it (`0x040b`) is replaced by the repeated part of the blue instruction (`0x0106`). The `ADD R5, R6, R3` instruction is skipped, and instead, the `ADD R3, R3, 0xa` instruction is repeated, since all of its 32 bits have been repeated. Finally, the `ADD R3, R3, 0xf` instruction is corrupted

as the top half of it (`0xf103`) is replaced by the repeated part of the `ADD R4, R4, 0xb` instruction (`0xf104`).

More on the ability to execute a new instruction

Since the encoding of the new Logical Shift Left instruction in Listing 3.11 is coming from the destination register and the second source operand in the `ADD R1, R1, 0x6` instruction, then changing these two operands to other values essentially enables to *"craft"* new instructions.

Table 3.2 shows examples of new instructions when changing these two operands. They have all been experimentally validated by clock glitch fault injection campaigns on both the Arm Cortex-M3 and Cortex-M4 devices. In other words, replacing the `0xf1010106` instruction from Listing 3.9 with an instruction from the first column of Table 3.2 enables to observe the execution of the corresponding instruction in the third column of Table 3.2. The second column illustrates the encoding of the new instruction, coming from the least-significant 16 bits of the original 32-bit instruction.

Original instruction	Least-significant 16 bits	New instruction
`ADD R4, R1, 0x9`	`0x0409`	`LSLS R1, R1, 0x10`
`ADD R0, R1, 0x46c`	`0x406c`	`EORS R4, R5`
`ADD R12, R1, 0x60c`	`0x6c0c`	`LDR R4, [R1, 0x40]`
`ADD R0, R1, 0x161`	`0x1061`	`ASRS R1, R4, 0x1`
`ADD R0, R1, 0x205`	`0x2005`	`MOVS R0, 0x5`
`ADD R3, R1, 0x416`	`0x4316`	`ORRS R6, R2`

Table 3.2: Possible 16-bit instructions coming from different destination registers and/or immediate value in the original 32-bit instruction.

These observations are particularly enlightening when it comes to previously observed fault models left unexplained. Trouchkine *et al.* examined a corruption of `R8` and `R0` when targeting a series of `AND R8, R8, R8` instructions [88]. They stated that corruption sometimes lead to a complete reset of the register. This `AND` instruction has the following encoding: `0xea080808`. Thanks to the proposed analysis, it is possible to fully explain the corruption they observed on the Arm Cortex-A53 processor, supporting the Thumb2 instruction set. In fact, the fault injection leads to the creation and execution of the 16-bit instruction `0x0808`. It is the encoding of `LSRS R0, R1, 0x20`.

This operation brings a reset of R0, since the value in R1 is shifted to the right by 32 bits and its result, obviously 0, is stored in R0.

Many instructions from Table 3.2 may lead to violate various security properties. For instance, executing an LDR (Load) instruction could reveal some values in the memory, breaking the confidentiality property. As another example, executing the EORS (XOR) instruction may allow an attacker to observe a collision in a cryptographic algorithm, which could lead to recover secret data. Or moving an immediate value to a register could result in corrupting a loop counter value if this register is used to store it.

It should also be noticed that most of the 16-bit instructions end with an **S**. This means that the APSR flags will be updated on the result of the instruction. Thus, executing new instructions that end with **S** explains why the APSR flags could be corrupted as a result of the fault injection, as already observed in chapter 2. Therefore, this could break the control-flow integrity of a program, in case these flags are going to be reused within the following instructions.

Furthermore, to show how powerful executing new instructions could be, and how the observed behavior depends on the original target encoding, another example is provided when targeting Listing 3.15. As a result of skipping the first 32 bits (0x0402f8d1), a totally new program is executed as shown in Listing 3.16. It is shown that two new instructions have been executed; one 32-bit instruction, and one 16-bit instruction

```
1 LSLS R2, R0, 0x10    // 0402f8d1
2 LDR R14,[r1,0xb00]   // eb00f103
3 ADD R3, R3, 0xe      // 030ebf00
4 NOP
```

Listing 3.15: Target program to exectue more than one new instruction.

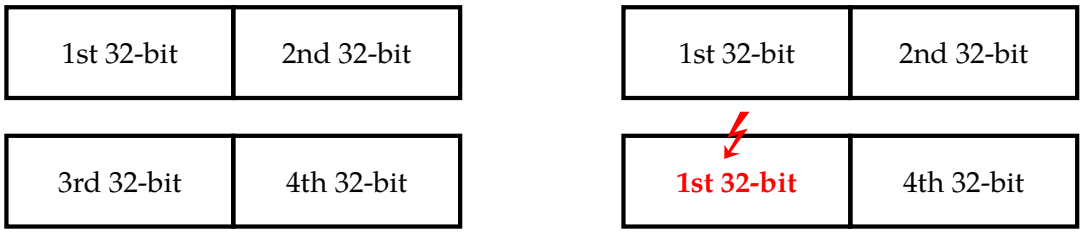
```
1 ADD R1, R0, R3, lsl #28  // eb00f103
2 LSLS R6, R1, 0xc         // 030ebf00
3 NOP
```

Listing 3.16: Observed exection for Skip 32 bits when targeting line 1 in Listing 3.15.

Additional faulty behaviors

This section gives examples of other observed faulty behaviors that can also be explained at the binary encoding level of the instructions, benefiting from the fact that the target ISA supports variable-length instructions. Nonetheless, these observed faulty behaviors depend on the target device.

Non-sequential skip and repeat 32 bits in Cortex-M4 device (STM32L4) This fault model is similar to the above-mentioned "Skip and repeat 32 bits". However, instead of skipping the 32 bits at line $i+1$, the 32 bits at line $i+2$ are skipped, while the 32 bits at line i are repeated. Figure 3.6 shows a possible faulty execution along with its golden one as a result of this specific faulty behavior.



(a) Golden execution. (b) Faulty execution.

Figure 3.6: Golden execution and observed faulty execution example as a result of "non-sequential 32-bit skip and repeat" in STM32L4.

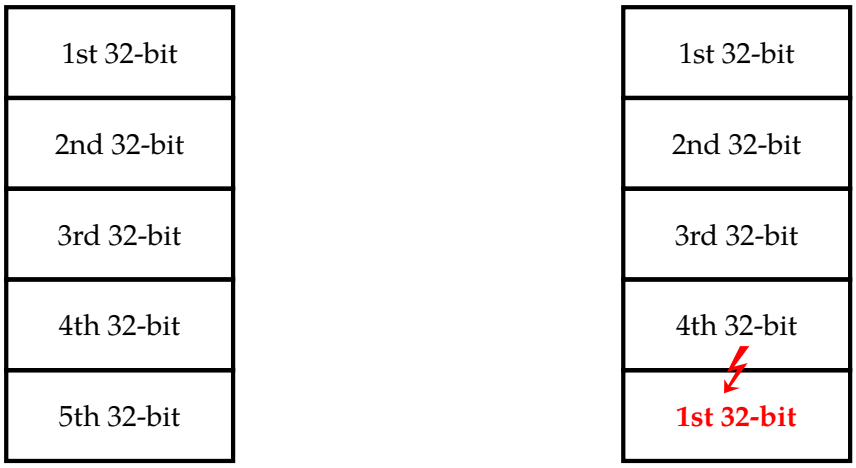
This behavior can be explained as a result of having 64-bit cache lines in the target device. Thus, each 64-bit cache line might be divided into two 32-bit chunks. Therefore, as a result of the injection, a 32-bit chunk can be affected by not updating its contents, resulting in repeating an already existing 32 bits and skipping another 32 bits, as shown in Figure 3.6. It should be noticed that the 32 bits can belong to any of the cases in Figure 3.4 and Figure 3.5.

Non-sequential skip and repeat 32 bits in Cortex-M3 device (STM32F1) Two kinds of this behavior are observed based on the order of the repeat:

- Out-of-order repeat: The repeated 32 bits are executed at the position of the skipped 32 bits. Nonetheless, only possible situation is observed for this behavior. It is when the distance between the repeated and the skipped bits are 96 bits, as shown in Figure 3.7. However, this does not mean that the 2nd 32 bits can replace

the 6th 32 bits based on this model. Instead the 5th 32 bits can replace the 9th 32 bits, and so on. A new fault model, defined in chapter 4, can explain this behavior as a result of a fault that affects the address of the fetched 32 bits in the Flash memory. As a result, the probability of observing this behavior depends on where the target code resides in memory.

- In-order repeat: The repeated 32 bits are repeated sequentially, regardless of the position of the skipped 32 bits. However, it was noticed that the maximum distance between the repeated 32 bits and the skipped 32 bits is 64 bits, excluding their bits, as shown in Figure 3.8.

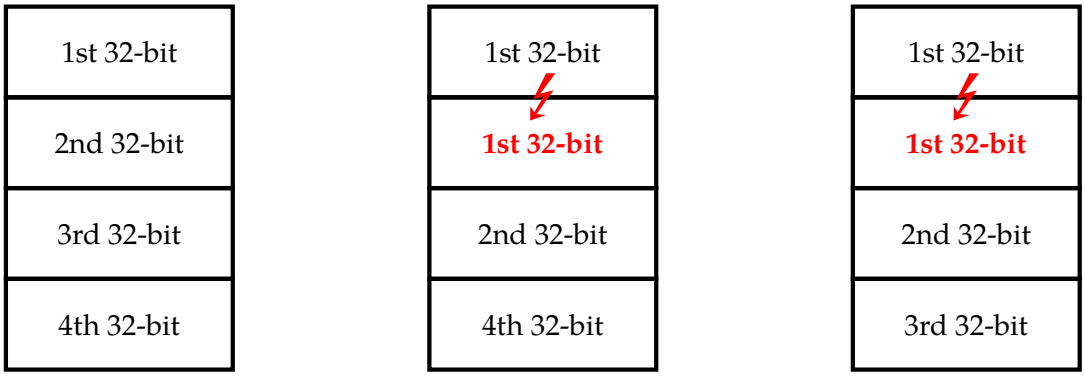


(a) Golden execution. (b) Faulty execution.

Figure 3.7: Non-sequential skip and repeat 32 bits in Cortex-M3 device (STM32F1) with out-of-order repeat.

It is worth mentioning that several injection campaigns on programs that allow the detection of these behaviors are performed, for both aligned and misaligned codes.

Some other faulty behaviors can be explained as *repeat an instruction*, or *skip and repeat the next instruction*, which is the opposite of the defined “Skip and repeat”. However, these two behaviors had a very low probability of occurrence, were not easy to reproduce, and were obtained when using a large value of shift: -49, and low values of width (*e.g.*, 1 or 3).



(a) Golden execution. (b) skip 3rd & repeat 1st. (c) skip 4th & repeat 1st.

Figure 3.8: Non-sequential skip and repeat 32 bits in Cortex-M3 device (STM32F1) with in-order repeat.

3.4 Exploitation and vulnerability analysis

In their work [119], the authors showed, through static analysis, that flipping some bits of variable-length instructions encoding could realign the code, resulting in dangerous erroneous behaviors. In addition, it was claimed in another paper [120] that variable-length instructions might bring more return-oriented programming (ROP) attacks. More on ROP attacks in [121].

This section presents a proof-of-concept example of how executing a new instruction can modify the `Program Counter`, resulting in breaking the control flow of the program. Moreover, this section provides a vulnerability analysis of implementations of the AES encryption algorithm using the preceding fault models, specifically when the code is misaligned in memory.

3.4.1 Program Counter modification

In this example, we leverage the ability to execute a new 16-bit instruction to control the `Program Counter`, where those 16 bits are originally belonging to a 32-bit instruction as already explained above. We consider this specific example, because controlling the `Program Counter`, as mentioned in [9], [79], could lead to harmful attacks, like privilege escalation or secure boot violation. Thus, the provided example works as a proof-of-concept for the potential exploitation of the presented results.

In the following, we assume that `R8` stores an address that points to a critical part of

the code, which can only be executed in a secure mode. Hence, our security property is to not modify the program counter to that critical address. This security property is violated if one manages to execute the `MOV PC, R8` instruction, for example. The 16-bit encoding of this instruction is `0x46c7`.

Several examples lead to violating this security property. Table 3.3 summarizes some of the instructions allowing that. Having one of these instructions in a misaligned code allows controlling the `Program Counter` to the value of `R8` when applying a fault injection attack. To validate this theoretical analysis, clock glitch fault injection campaigns were performed on the code in Listing 3.17.

Original instruction	Least-significant 16 bits
`ADD R6, R1, 0x4c7`	`0x46c7`
`SUB R6, R1, 0x4c7`	`0x46c7`
`MOVW R6, 0x4c7`	`0x46c7`
`LDR R4, [r0, 0x6c7]`	`0x46c7`
`ORR R6, R6, 0x63800000`	`0x46c7`

Table 3.3: Instructions that lead to modify the PC to the value in R8 when performing clock glitch fault injection.

```
1  //prologue
2  MOVW R8, 0x056e // storing the critical-
3  MOVT R8, 0x0800 // -address in R8
4  //series of NOPs
5  LSLS R2, R0, 0x10
6  //any instruction from Table 3.3
7  ADD R3, R3, 0xa
8  ADD R4, R4, 0xb
9  ADD R5, R6, R3
10 ADD R3, R3, 0xf
11 //series of NOPs
12 LDR R1, [R1,0xf00]
13 MOV R9, R6
14 //epilogue
```

Listing 3.17: Target program for `Program Counter` modification experiments.

A specific address is stored in `R8` in the Prologue. This address refers to line 12 in Listing 3.17. The instruction at line 6 can be any instruction from Table 3.3, and we

were able to validate all of them. In other words, we were always able to jump to line 12 in Listing 3.17 when performing the attack on any of the instructions in Table 3.3.

Other instructions may be a source for such vulnerability, especially if other source registers store critical addresses, and not only `R8` as in this example. However, most of the original instructions that could be a source of such vulnerability have `R6` as a destination register, as already shown in Table 3.3. Gratchof *et al.* [122] said that there is a higher probability to change the value of the program counter when the destination register of a `MOV` instruction is `R6`. They observed various jumps in a program when performing fault injection attacks on a Cortex-A9, which supports two execution states: Thumb2 and ARM32. Hence, these observations can reasonably explain their experimental results regarding the `R6` register, assuming that their execution state was Thumb2.

3.4.2 Vulnerability analysis of AES implementations

This study focuses around three different AES implementations: *BroAES*[1], *TinyAES128*[2] and *MbedTLS-AES*[3]. Such tests illustrate the real-life applicability of misaligned faults. First, we concentrate on an exploitable branch logic error within *BroAES*, experimentally validated by clock glitch fault injection. Then, we cover two general observations with examples from *TinyAES128*, then *MbedTLS-AES*.

AES is a round-based symmetric encryption algorithm. It iteratively transforms an input plaintext into a ciphertext. The last round differs slightly from the previous ones, omitting the *MixColumns* transformation and adding an extra *AddRoundKey*. Consequently, software implementations of the algorithm usually loop over the iterations, while making an exception for the last round. The first section targets this branch logic.

It must be mentioned that the work in this section was done in collaboration with **Gijs Burghoorn** while he was an intern at TIMA.

Branch logic error on *BroAES*

A clock glitch attack on *BroAES* is depicted here, exposing the key in a “known plaintext” scenario with the use of the described fault models.

1. `https://github.com/brobwind/bro_aes`
2. `https://github.com/kokke/tiny-AES-c`
3. `https://github.com/Mbed-TLS/mbedtls`

In *BroAES*, all rounds are encapsulated in a `for` loop. Depending on the round, some transformations are excluded. Listing 3.18 shows the instructions initializing the memory, used to keep track of iterations and branching.

```
1 |b087|e9d0  // SUB   SP, 0x1c
              // LDRD  R7, R8, [R0, 0x8]
2  7802|f8d2  // LDR.W R0, [R2, 0xb0]
3  00b0|9001| // STR   R0, [SP, 0x4]
              // [SP+4] = num_rounds
4 |3801|9002| // SUBS  R0, 0x1
              // STR   R0, [SP, 0x8]
              // [SP+8] = num_rounds-1
5 |2000|f102  // MOVS  R0, 0x0
              // ADD.W R4, R2, 0xb4
6  04b4|9000| // STR   R0, [SP, 0x0]
              // [SP+0] = iteration count
7 |e9dd c000| // LDRD  R12, R0, [SP]
8 |4560|da06| // CMP   R0, R12
              // BGE.N <...>
```

Listing 3.18: Initialization of memory used for iteration.

Before the instructions at line 8, checking the loop condition and branch to the transformations, there are three `STR` instructions pushing values onto the stack. Line 3 causes `[SP+4]` to contain the value of `num_rounds`. Later instructions use it to verify the loop condition and determine whether to perform the *SubBytes* and *ShiftRows* transformations. Line 4 causes `[SP+8]` to contain the value of `num_rounds - 1`. Later, the loop body uses it to assess whether to perform *MixColumns* transformation. Line 6 makes `[SP+0]` contain the loop iteration value.

Targeting the instruction `LDR.W R0, [R2, 0xb0]` at line 2, with the fault model described above in order to skip 32 bits at line 3, leads to what is reported in Listing 3.19:

The value of `[SP+4]` not being set, it is therefore an arbitrary value. Since no earlier instruction touches this part of the stack, one can reasonably assume that in the first round `[SP+4]` is set to `0`. The value of `[SP+8]` is set to `R0` which is initially a pointer to a stack value. The value of `[SP+0]` is handled normally and set to `0`. Then, after executing the `LDRD` instruction at line 6, `R12` is set to `[SP+0]`, which is `0` and `R0` is set to `[SP+4]`, which is also `0`. This causes the `CMP` instruction at line `7` to compare `0` with `0`. As a result, the loop only runs once. The flags set by the `CMP` instruction are reused

within the loop body to select transformations.

At a higher level, the effect is a such: First, the loop body performs the *AddRoundKey* transformation, then it skips *SubBytes* and *ShiftRows* transformations because the flags set by CMP are reused; lastly, it performs *MixColumns* transformation. However, since the loop is not executed any more, the resulting value of this transformation is never used again.

```
1 |b087|e9d0  // SUB   SP, 0x1c
              // LDRD  R7, R8, [R0, 0x8]
2  7802|f8d2  // LDR.W R3, [R2, 0x801]
3  3801|9002| // STR   R0, [SP, 0x8]
4 |2000|f102  // MOVS  R0, 0x0
              // ADD.W R4, R2, 0xb4
5  04b4|9000| // STR   R0, [SP, 0x0]
6 |e9dd c000| // LDRD  R12, R0, [SP]
7 |4560|da06| // CMP   R0, R12
              // BGE.N <...>
```

Listing 3.19: Observed execution for Skip 32 bits / single instruction skip and single instruction corruption when targeting line 3 in Listing 3.18.

In the end, only the *AddRoundKey* transforms the output ciphertext. Therefore, in a known plaintext scenario, Equation (3.1) holds and allows to recover the secret key.

$$\begin{aligned} ciphertext &= plaintext \oplus key \\ key &= ciphertext \oplus plaintext \end{aligned} \tag{3.1}$$

Clock glitch campaigns on Arm Cortex-M3 device (STM32F1) and Cortex-M4 device (STM32L4) have confirmed the presence of this behavior beyond theoretical analysis. It should be mentioned that the “Skip 64 bits” fault model could also lead to the same exploitation. On both devices, this exploitation is highly reproducible.

Early return on *TinyAES128*

This section illustrates a possible future attack vector with the described fault models, allowing for an Early Return from a function. An excerpt from *TinyAES128* containing the necessary instruction pattern is detailed here.

The new fault models enable the exploitation of consecutive branch instructions. In *TinyAES128*, the encryption function utilizes separate functions to perform various encryption transformations. The encryption function then calls these functions sequentially. Listing 3.20 highlights a part of the resulting instructions. When these calls use misaligned 32-bit `B` or `BL` instructions, the fault models allow attackers to turn branches into Early Returns, giving way to return from the encryption function itself before performing the transformations.

After an Early Return, the memory for the resulting ciphertext contains an internal state used during the encryption process. A similar attack to the one described in *BroAES* can then be mounted.

```
1 ....|f7ff  // BL <SubBytes>
2 ffd1|f7ff  // BL <ShiftRows>
3 ff9d|....
```

Listing 3.20: Misaligned consecutive branch instructions within TinyAES128.

Thumb2 branch instructions use a *relative* instruction offset. The 32-bit branch instruction encoding stores the 12 least-significant bits of this offset in the second-half (16 bits) of the encoding [77]. Because of these two points, applying the "Skip 32 bits" model on the first of two consecutive misaligned 32-bit branch instructions effectively ignores the first branch and executes the second branch offset by -32 bits as shown in Listing 3.21.

Usually, 32 bits before a function is the `POP`/`return` instruction of the calling function. Since no `PUSH` instruction was given beforehand, the result is an Early Return from the calling function, which is the whole encryption function in our case.

Certain conditions apply on functions to witness this behavior: for example, the position of the function declarations, the similarity of function parameter types, and the usage of the link register. It is also important to consider that not all registers might be properly restored when returning. This is especially true for the `BL` instruction.

```
1 ....|f7ff  // BL <ShiftRows-0x4>
2 ff9d|....
```

Listing 3.21: Observed execution for Skip 32 bits / single instruction skip and single instruction corruption when targeting line 2 in Listing 3.20.

Hint and control instructions in *MbedTLS-AES*

This section describes the usage of hint and control instructions to mimic instruction skips. This would simplify conducting DFA attacks on this implementation of AES.

The Thumb2 instruction set contains hint and miscellaneous control instructions [77]. They detail a requested internal behavior to the processor concerning memory usage (*e.g.* `PLD`), system events (*e.g.* `WFI`), speculative execution (*e.g.* `CSDB`) and pipelining (*e.g.* `ISB`). In most places, these instructions have no impact on execution and behave like a `NOP` within the execution flow. This makes them of special interest when exploiting the portrayed instruction corruption fault models.

Within the explored targets, instruction corruption only ever yields `PLD` hint instruction. Within *TinyAES128*, `PLD` instruction has undefined arguments and is therefore unusable. In *MbedTLS-AES*, both the `-O1` and `-O2` compiler optimization levels yield corrupted instructions to precise `PLD` instructions. Listing 3.22 and Listing 3.23 show how the `LDRB.W` instruction has been corrupted to a `PLD` one.

```
1  ....|fa53  // UXTAB LR, R3, LR
2  fe8e|f89e  // LDRB.W R2, [LR, #40]
3  2028|69f6| // LDR R6, [R6, #28]
4 |4072|....  // EORS R2, R6
```

Listing 3.22: Selection of MbedTLS-AES encryption instructions.

```
1  ....|fa53  // UXTAB LR, R3, LR
2  fe8e|f89e  // PLD   [LR, #3726]
3  fe8e|f89e  // LDRB.W R4, [LR, #114]
4  4072|....
```

Listing 3.23: Observed execution for Skip & repeat 32 bits / double instruction corruption when skipping line 3 and repeating line 2 in Listing 3.22 .

Further remarks

An important point to be noticed is that any exploitation using the described fault models is extremely sensitive to minor choices made by the compiler and linker. The optimization levels, positions of functions and chosen instructions encoding play a major

Target code	Optimization level	Inserted faults	Undefined instructions
BroAES	`-O0`	29	10 (34.5%)
	`-O1`	59	16 (27.1%)
	`-O2`	87	31 (35.6%)
	`-Os`	85	19 (22.4%)
TinyAES128	`-O0`	33	11 (33.3%)
	`-O1`	44	21 (47.7%)
	`-O2`	106	17 (16.0%)
	`-Os`	52	26 (50.0%)
MbedTLS-AES	`-O0`	69	35 (50.7%)
	`-O1`	293	84 (28.7%)
	`-O2`	283	60 (21.2%)
	`-Os`	193	52 (26.9%)

Table 3.4: Number of possible fault insertions with 32-bit misaligned instruction corruptions within the encryption function and the number of created undefined instructions for the explored target codes at several optimization levels.

role in deciding whether these fault models would produce exploitable behaviors. Furthermore, many applications of the misaligned instruction corruption fault models lead to *undefined* instructions which would cause a crash or a fault handler to trigger. An indication of the frequency of these undefined instructions for the different implementations of AES is reported in Table 3.4. Other applications can create *unpredictable* instructions which may behave differently across architectures and activate fault handlers too.

Even after considering all these comments, every target and optimization level still creates numerous possible injection points. Therefore, it is common to find multiple injection points that execute without crashing and where changes to the output or control flow of the program can be witnessed. Whether these injection points are fully exploitable depends on the target program and the target architecture.

3.5 Fault models simulation

In order to simulate the inferred fault models and be able to analyze the outcomes of larger fault injection campaigns, a *Software simulator and analyzer* automation tool

was developed by **Oumayma Teyeb** during her internship. This internship was done at LCIS lab under my supervision. This tool is available online on GitHub[4].

The main functionalities of the tool are as the following:

- Emulate a Cortex-M processor so that the tool can run any assembly program written using the Thumb2 instruction set.
- Simulate "Skip" and "Skip and repeat" for a specific number of bits fault models, either with aligned or misaligned code. The tool considers various numbers of bits. This includes, for example, 32 and 64 bits.
- Simulate the "non-sequential skip and repeat" fault model for aligned codes.
- Automate the comparison between the outcomes of software fault models simulation and the outcomes of physical fault injection or RTL fault simulation. The comparison is performed based on the values of the general purpose registers.
- Provide statistical analysis of the percentage for the silent cases, the crashes, the faulty behaviors that have defined fault models (*i.e.* Coverage), and the ones that do not have known fault models.
- Generate a PDF report of the analysis and the comparison results.

It should be mentioned that the tool has some limitations for misaligned codes. This is because the simulation was mostly based on the assembly level and not the binary encoding level. More details and how the tool can be installed and used are available on the GitHub link.

3.6 Fault models evaluation

Metrics from section 2.3 (*i.e.* Coverage, Fidelity, and Complexity) are used to evaluate the aforementioned fault models. Extensive clock glitch fault injection campaigns, using four different target devices, were performed while executing the target program shown in Listing 3.24. The experimental parameters of these campaigns are illustrated in Table 3.5. Three different delay values are used to target different locations within the target program. These delay values depend on the number of initialization instructions

4. `https://github.com/oumTe/fault_model_simulator`

in the Prologue, and the number of `NOPs` between the Prologue and the Target parts. Although the same target program is used for all devices, different delay values had to be used for STM32F4 to observe faulty behaviors. This can be explained as this device has additional hardware components. Repetition is the number of executions for each combination of parameters. For each fault injection campaign on a target device, the total number of experiments corresponds to 750 000 executions, as summarized in the last row of the table. It should be mentioned that only the STM32F1 embeds an Arm Cortex-M3 core, while the rest embeds an Arm Cortex-M4 core.

```
1 ADD  R1, R1, 0x6
2 ADD  R3, R3, 0xa
3 ADD  R4, R4, 0xb
4 ADD  R5, R5, 0x1
5 ADD  R2, R6, 0xd
6 ADD  R3, R3, 0x9
7 ADD  R6, R6, 0x4
8 ADD  R2, R2, 0x3
```

Listing 3.24: Target part for fault models evaluation campaigns.

	Target device			
	STM32F1	STM32F3	STM32L4	STM32F4
Shift	[-49,0]			
Width	[0,49]			
Delay	{38, 39, 40}			{63, 64, 65}
Repetition	100			
Total	750 000			

Table 3.5: Experimental setup for fault models evaluation experiments.

Table 3.6 shows the results in terms of Silent, Crash, and Faults. Additionally, it shows the Coverage percentage within the observed faulty behaviors. It is shown that the Coverage is very high for almost all devices. This shows how the inferred fault models are useful, and they were able to cover most of the obtained faults. It is worth mentioning that the described tool in section 3.5 was used for this analysis and computing the Coverage percentage.

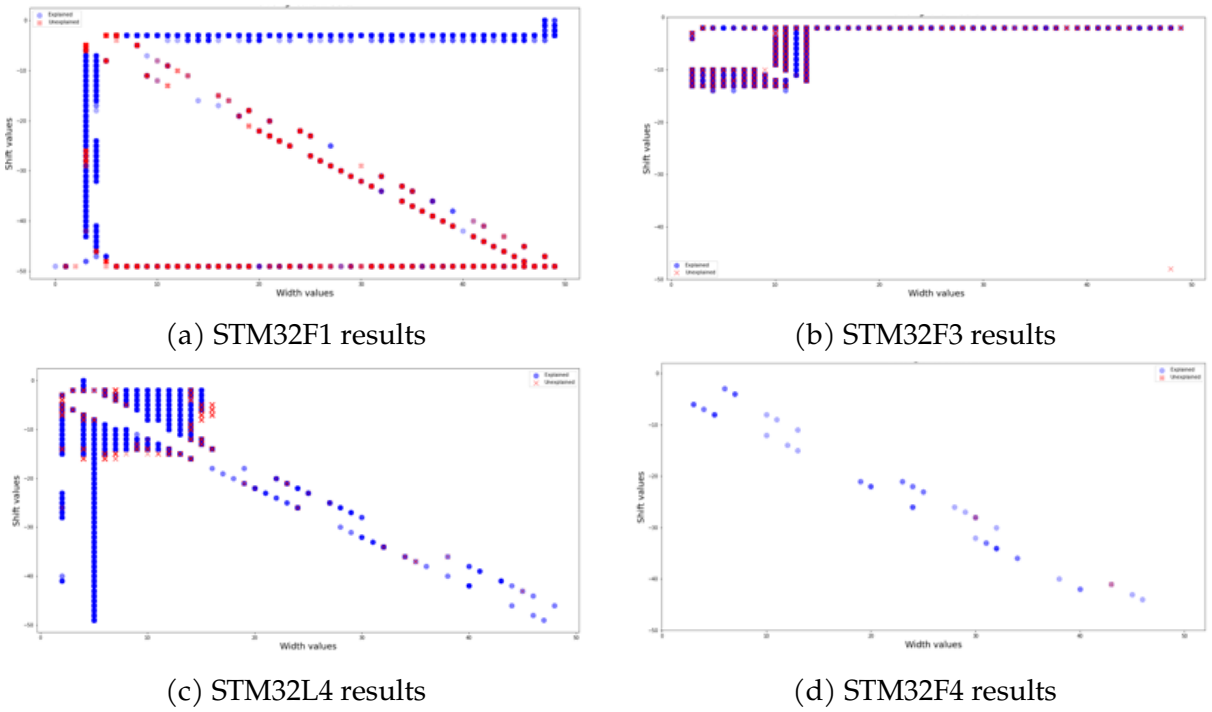
Figure 3.9 illustrates the distribution of the obtained faults with respect to the shift values (y-axis), and the width values (x-axis) of the glitch. Blue circles ● mean the observed fault has a corresponding known fault model, while the red x corresponds to a fault that has an unknown fault model. The darkness of the color represents the repetition of the fault at the corresponding combination of shift and width. One can conclude from the figures that specific regions of shift and width lead to faulty behaviors. In addition, specific regions of shift and width lead to explained or unexplained faulty behaviors (*i.e.*, have known or unknown corresponding fault models). This is obvious in Figure 3.9c, for example.

	Target device			
	STM32F1	STM32F3	STM32L4	STM32F4
Silent	96.77	96.84	96.4	95.2368
Crash	0.57	1.95	0.36	4.7557
Fault	2.66	1.21	3.24	0.0075
Coverage	90.16	73.05	93.28	96.43

Table 3.6: Experimental results for fault models evaluation experiments, values in %.

On the other hand, as the occurred faulty behaviors that are classified under “Skip” and “Skip and repeat” fault models are already predicted, then Fidelity equals 100 % for these fault models. To put it in another way, as these fault models are realistic and can explain observed faulty behaviors from physical injection, then the Fidelity is 100 %. However, it was noticed that one model could be more probable than the other for a device. For example, “Skip” is observed more than “Skip and repeat” in STM32F3, while “Skip and repeat” appeared more than “Skip” in STM32F1. Nevertheless, in most cases, tuning the glitch parameters would be able to increase the probability of a faulty behavior. For “non-sequential skip and repeat”, the Fidelity depends on the target device. For instance, for “non-sequential skip and repeat in STM32L4”, the Fidelity is 100 %. However, the Fidelity of observing this model on another device, such as STM32F3, is 0 %. But this is negligible, as it is already not expected to obtain this kind of faults when targeting STM32F3. This is because STM32F3 does not support cache lines, as in the case of STM32L4.

Regarding the Complexity metric, it would be easy to perform vulnerability analysis even manually if the code is aligned. Conversely, it would be more difficult when the



(a) STM32F1 results

(b) STM32F3 results

(c) STM32L4 results

(d) STM32F4 results

Figure 3.9: Explained and unexplained faults classification with respect to shift and width values of a glitch. Y-axis: Shift $\in$ [-49,0], X-axis: Width $\in$ [0,49].

code is misaligned. Nonetheless, having an automation tool to analyze a code at the binary encoding level regardless of its alignment would certainly make the process easier. Similarly, for countermeasures, as the fault models are understandable, the design or developing process of countermeasures should be easy, in particular, when security properties are well-defined, or critical parts within the code are well recognized. Thus, the focus will be on what kind of vulnerabilities can be exploited as a result of applying the fault models, and then what kind of modification to the code should be applied to prevent these vulnerabilities from getting exploited, for example, by choosing or avoiding specific registers. Furthermore, the fault models are *explainable*, and their realism is proved. For their description, they are clear, understandable and can be simulated easily on a given piece of code. Finally, the previous sections showed that “Skip” and “Skip and repeat” are applicable to different target programs and target devices. They are therefore independent of the target program and device, which prevents limiting them in certain conditions. Conversely, it is clear that this is not the case for more complex fault models like “non-sequential skip and repeat”.

3.7 Conclusion

This chapter is introduced by a series of hardware fault simulation experiments that provided knowledge on the origin of some of the observed faulty behaviors at the microarchitectural level. More importantly, these experiments revealed the rationale behind different faulty behaviors which were observed when applying a small modification to the target program; this is due to the Thumb2 instruction set supporting variable-length instructions, which can lead to aligned or misaligned code in memory. Thus, the processor might fetch aligned or misaligned instructions. This was the prime knowledge to infer new fault models at the binary encoding level of the instructions. The main inferred models are “skip” and “skip and repeat” a specific number of bits. This number of bits is related to the flash memory access size, cache line size, instruction cache, or register size in the fetch path. Clearly, these models offered explanations for most of the observed faulty behaviors. Moreover, their realism was proved by targeting different instructions and using various target devices. Additionally, exploitation and vulnerability analysis examples using the proposed models were performed and validated experimentally. Finally, this chapter described an automation tool to simulate these models and perform the comparison between the different injection and simulation results.

4

Hardware fault simulation and partial update fault model

This chapter moves downward at system level and considers the hardware level in more detail, to perform hardware fault simulation experiments. This enables a better understanding of the faults' propagation, validates the inferred fault models at ISA and binary encoding levels, and reveals the origin of such faults at microarchitectural level. These simulation experiments have led to the inference of a novel new fault model. Many of the reported faulty behaviors that were classified under unexplained faults, in the previous chapter, can be now understood using this new fault model.

In the following, section 4.1 describes the fault simulation at hardware level, along with the followed approach in RTL fault simulation, the proposed RTL fault models, and additional observations and conclusions that validate the use of the proposed fault models at different levels of abstractions. Section 4.2 presents the *partial update* fault model, along with the associated results, analysis, and conclusions. Section 4.3 evaluates the proposed models. The chapter is concluded in section 4.4.

4.1 Hardware fault simulation

To better understand and pinpoint the origin of the faulty behaviors described in section 3.3, RTL fault simulation experiments have been performed on the same target programs used in chapter 3. This should also confirm the observations regarding the executed instructions. The RTL descriptions used in the simulation are for both the Arm Cortex-M3 and Arm Cortex-M4 processors.

It should be mentioned that these simulation experiments were performed after the AAA agreement with Arm was accomplished. Therefore, these descriptions have non-obfuscated RTL description for the core itself. However, they are related to a single integrated system that includes the core. Thus, the whole system works with a single main clock, which is not similar to the DesignStart version (described in chapter 3), where the RTL description consists of subsystems.

The following subsections provide the chosen methodology of RTL fault simulation, the RTL fault models permitting to observe the same faulty behaviors at higher levels, and explanations of these fault models using post-synthesis clock glitch simulation.

4.1.1 RTL fault simulation methodology

With RTL fault simulation, the goal is to find the signals or internal registers[1] that, when faulty, lead to the erroneous behaviors described in the previous chapter, and hence, to reveal their origin and better understand their propagation at lower levels of abstraction. However, since there are thousands of registers in the RTL description of a processor such as the Arm Cortex-M3, a specific methodology is necessary to accelerate the analysis and find the target registers in a reasonable time. To this end, path delay analysis within specific architectural components or modules has been set up. Clock glitch being likely to cause timing errors, then critical or almost critical paths are more inclined to be faulty [35], [123], [124]. Consequently, the registers involved in such paths have more chances to capture faulty values as a result of path timing violations.

Based on the above, the proposed approach consists in faulting the involved registers in the paths that come first, according to the maximum paths delays, when generating a timing analysis report for specific architectural components or RTL modules. The destination and source registers, of these paths, are the involved registers. The fault models detailed in subsection 4.1.2 are then applied on them. A divide and conquer approach is used to perform the simulation on an architectural component at a time. Vivado environment has been used to generate the timing analysis reports, while QuestaSim has been used to conduct the RTL fault simulation.

Algorithm 1 describes the followed approach in details. It takes three lists as input:

- the RTL modules which compose the whole RTL description or a subset of it for

1. The word "register" here does not refer to a purpose register, but to one or multiple D-flip flops that store an internal value.

a given processor,

- the RTL fault models that are used to perform the RTL fault simulation,
- the experimental results of the clock glitch fault injection.

Additional input is N, which is the maximum number of paths that are going to be generated from the timing analysis report. The output of the algorithm is a list of the registers R that cause faulty behaviors, that are identical to those obtained by clock glitch. Table 4.1 provides descriptions for the functions that are used in Algorithm 1.

Algorithm 1: RTL fault simulation methodology

```
Input: rtl_modules, rtl_models, injection_results, N.
Output: R.
modules ← rtl_modules;
models ← rtl_models;
inj_res ← injection_results;
R ← [ ];
while !isEmpty(modules) do
    m ← pop(modules);
    path_list ← timingReport(m, N);
    while !isEmpty(path_list) do
        p ← max(path_list);
        delete(p, path_list);
        dst ← getDestination(p);
        src ← getSource(p);
        while !isEmpty(models) do
            model ← pop(models);
            rtl_res ← getRegValues(dst, model);
            if isEqual(rtl_res, inj_res) then
                R ← R ∪ dst;
            end if
            rtl_res ← getRegValues(src, model);
            if isEqual(rtl_res, inj_res) then
                R ← R ∪ src;
            end if
        end while
    end while
end while
return R;
```

Function	Description
`pop(l):`	returns and removes the first element from list `l`.
`delete(e, l):`	delete element `e` from list `l`.
`timingReport(m,n):`	returns list of `n` paths of maximum delay in module `m` using Vivado. Each path consists of its source, destination, and delay.
`max(l):`	returns the path with maximum delay in list `l`.
`getDestination(p):`	returns destination register from path `p`.
`getSource(p):`	returns source register from path `p`.
`getRegValues(r,m):`	returns the values of registers `R0` to `R12` after performing RTL fault simulation on register `r` with fault model `m` using QuestaSim.
`isEqual(a,b):`	returns true if list `a` is identical to list `b`.
`isEmpty(l):`	returns true if list `l` is empty.

Table 4.1: Algorithm 1 description.

4.1.2 RTL fault models

The aforementioned methodology has been applied to the RTL modules that are relevant to the *fetch* stage of the processor, in order to confirm our assumption, in the previous chapter, regarding the origin of the observed faulty behaviors.

To implement RTL fault simulation, two fault models have been introduced. They follow the intuition behind timing violations, being that the value of a register might not be correctly updated.

The first RTL fault model consists in *preventing the update* of a register value at a given clock cycle. Therefore, the register keeps its previous value. This RTL fault model validates the “Skip and repeat 32 bits” faulty behavior at binary encoding level and explains it at RTL level. All the registers located in the fault propagation path, shown in Figure 4.1, have generated the same faulty behavior; meaning the faulty register can either be in the interface between the fetch unit and the AHB, in the AHB component, or in the interface between the AHB and the flash memory. The propagation starts from the interface between the fetch unit and the AHB: This interface and the fetch unit are parts of the fetch microarchitectural component, which is part of the core itself. Here, the fault propagation follows the opposite direction of the instruction data path from the flash memory to the processor core, *i.e.,* the *fetch* path. Then, the fault propagation continues from the flash memory to the fetch unit. Thus, the origin faulty register could

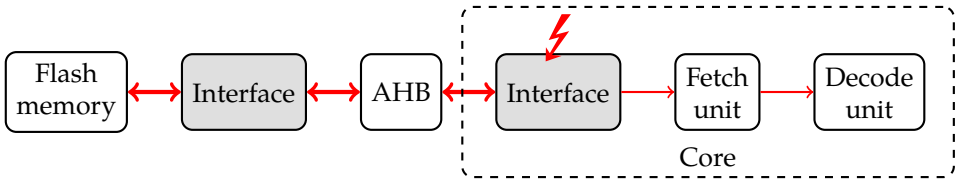


Figure 4.1: Fault propagation path for Skip 32 bits or Skip & repeat 32 bits fault models.

be in the opposite path of the *fetch*, or in the same path as the *fetch*.

The second fault model involves *anticipating the update* of the value of a register at a given clock cycle; at clock cycle i, the value that the register would actually store at clock cycle $i + 1$ is loaded. This RTL fault model leads to the "Skip 32 bits" behavior at binary encoding level. Fewer registers, when targeted, generate the "Skip 32 bits" behavior, and all are located in the interface between the fetch unit and the AHB as shown in Figure 4.1.

4.1.3 Post-synthesis timing simulation

In order to explain how the aforementioned RTL fault models have validated the same faulty behaviors observed at higher levels of abstraction, another layer has been taken into account: performing post-synthesis clock glitch simulation on an FPGA using Vivado. There again, the implementation of this simulation has targeted modules related to the "fetch" stage of the processor. The objective of this test has been to investigate on the effects of the clock glitch over specific registers within these modules from the architectural perspective.

As a result, three cases have emerged, as presented in Figure 4.2:

- *Additional cycle*: the glitch acts as an extra clock cycle. In other words, register `R` is normally updated because of the glitch.
- *Silent*: the glitch has no effect on the values of `R`. However, it has been observed that `R` is updated at the rising edge of the glitch and not at the rising edge of the following clock cycle. Hence, the final value is not affected.
- *Fault*: the effect is same as *Silent* or *Additional cycle*, however, faulty values have been monitored at the rising edge of the clock cycles following the glitch, either the first rising edge or the ones after, as a result of a timing violation. Various faulty values are captured based on fine-tuning of the glitch parameters. Among

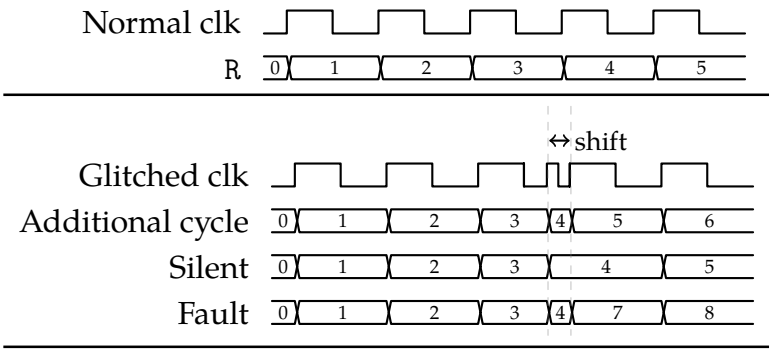


Figure 4.2: Possible effects of post-synthesis clock glitch simulation on register R.

them, we could observe that the value is same as a value occurred two clock cycles earlier, two cycles later, some are incoherent values , some bits are reset. And more.

The glitch parameters are the reason behind these three different effects. It has to be remembered that the effect of the glitch, with given parameters, might not be the same for all the registers within the same module; the reason being that not all paths between registers have the same delay. Based on that, it's possible to observe faulty behaviors due to the various effects on the registers. Therefore, by considering the glitch effect on the output of source and destination registers, Figure 4.3 and Figure 4.4 report how the "Skip" and "Skip and repeat" behaviors can come up. The effect of the glitch is similar on the source and the intermediate register in both figures: the source register is subject to *Additional cycle* effect, while the intermediate undergoes *Silent* effect. However, in the second clock cycle that follows the glitch, the intermediate register captures the faulty value 4, the one available at that time. Thus, the glitch effect on the destination register determines whether the resulting erroneous behavior would be a "Skip" or "Skip and repeat". In Figure 4.3, the destination register is subject to *Silent* effect, and hence, no value is repeated. Therefore, the obtained effect is "Skip". On the other hand, in Figure 4.4, the destination register takes *Additional cycle* effect, which leads to capture the value 2 twice. Thus, in this case, a "Skip and repeat" behavior is procured. In addition to that, by combining more registers, "Non-sequential skip and repeat 32 bits with in-order repeat" can be explained. For example, Figure 4.5 shows how "Skip 5 and repeat 2 " may occur. All the intermediate registers are subject to *Silent* effect, while the destination register is under the *Additional cycle* effect. All these detailed observations and conclusions validate the use of the aforementioned

fault models at higher levels.

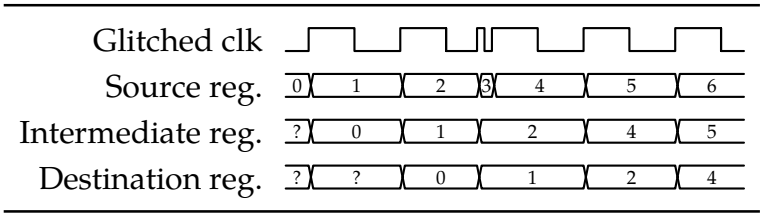


Figure 4.3: "Skip" behavior description with timing simulation: Skip 3.

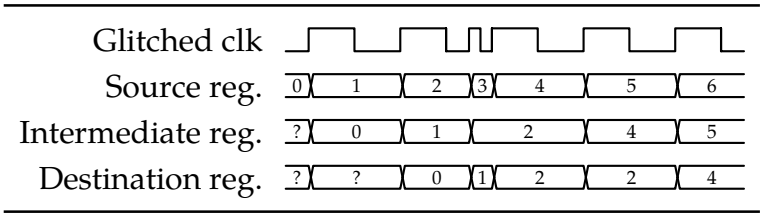


Figure 4.4: "Skip and repeat" behavior description with timing simulation: Skip 3 and repeat 2.

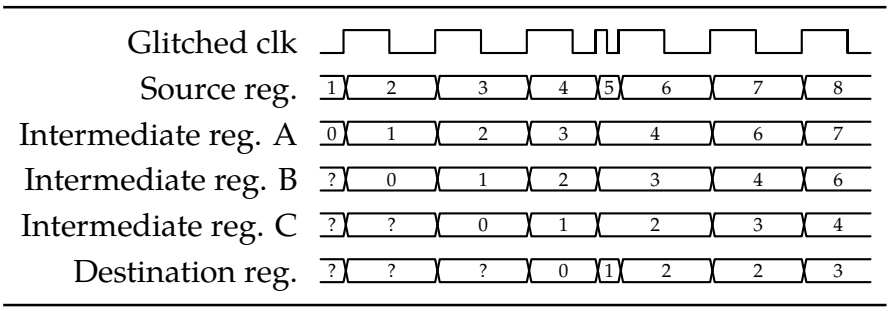


Figure 4.5: "Non-sequential skip and repeat 32 bits with in-order repeat" behavior description with timing simulation: Skip 5 and repeat 2.

Figure 4.6 shows the result of applying RTL fault models on registers. In order to model "Skip and repeat" at RTL level, *preventing the update* fault model is applied to either the intermediate or the destination register or even any further destination register; this results in getting two consecutive 2s instead of 2 then 3. On the other hand, aiming for "Skip" requires *anticipating the update* fault model applied at the source register; thus 4 appears instead of 3 after 2.

It is worth mentioning that applying *preventing the update* fault model on the source register would lead to have *repeat only* faulty behavior, which is already observed in

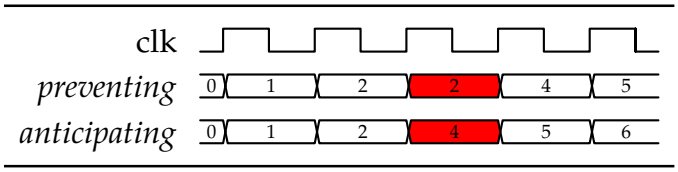


Figure 4.6: Result of applying RTL fault models.

very few cases as mentioned in chapter 3. On the other hand, applying *anticipating the update* fault model on the intermediate or destination registers would lead to observe *skip the previous and repeat the next*, which is also observed with low probability and less reproduciblity. Finally, simulating “Non-sequential skip and repeat 32 bits with in-order repeat” requires combining both models. For instance, to simulate “Skip 5 and repeat 2", *preventing the update* is applied on the source register at clock cycle i to repeat 2, then at clock cycle $i+3$; *anticipating the update* is also applied on the source register to skip 5.

4.1.4 Summary

These simulation experiments have confirmed our observations on executed instructions in the previous clock glitch experiments. Therefore, they validate the inferred 32-bit fault models at binary encoding level, in particular, “Skip 32 bits” and “Skip and repeat 32 bits”. They also corroborate our assumption on the origin of such faulty behaviors: the *fetch* stage in the pipeline. Additionally, they have unfolded the discriminating rationale behind the “Skip” and “Skip and repeat” fault models. We are confident that performing identical simulations on RTL description with 64-bit cache-line size would unravel the corresponding 64-bit fault models as well.

4.2 Partial update fault model

Until now, the RTL fault simulation has primarily focused on individual internal registers as the fundamental element. However, this approach raises a pertinent question: what if a fault only affects a subset of flip-flops within a register, rather than impacting all of them? The motivation behind this query stems from the fact that the update of flip-flops within a register may not necessarily occur simultaneously.

In order to address this question, a reverse analysis process was conducted on the

results of the unexplained faulty behaviors. This process ultimately led to the inference of a new fault model: the *partial update* fault model.

In the following, subsection 4.2.1 shows examples that allowed inferring the new model with its sub-cases. These sub-cases are defined in subsection 4.2.2. The experimental result of each sub-case are then illustrated in detail in subsection 4.2.3 and subsection 4.2.4.

4.2.1 Inference examples

While trying to analyze and understand the unexplained faulty behaviors, it has been noticed that the obtained faulty values of the general purpose registers can be explained as a result of executing instructions that have a partially correct binary encoding. Conversely, the remaining incorrect part is belonging to either zeros or to values that are identical to the encoding of the last previously fetched data. As an example, Listing 4.1 shows a target program, which is the same as the target used in section 3.6. Listing 4.2 shows how one of the unexplained faulty behaviors can be explained at ISA and binary encoding levels. The new faulty execution can be seen as resetting 6 bits (indices: 0, 1, 28, 29, 30 & 31) of the original encoding at line 3 in Listing 4.1 (`0xf104040b`). As a result, two new 16-bit instructions are executed: `0x0104`, then `0x0408`. There are several other examples that have also been explained with the same observation, which count almost for all of the unexplained faulty behaviors, as will be illustrated in section 4.3. Additionally, this is applicable to faults either in 32 bits or 64 bits, as will be detailed in subsection 4.2.3 and subsection 4.2.4.

```
1 ADD  R1, R1, 0x6
2 ADD  R3, R3, 0xa
3 ADD  R4, R4, 0xb   // 0xf104040b
4 ADD  R5, R5, 0x1
5 ADD  R2, R6, 0xd
6 ADD  R3, R3, 0x9
7 ADD  R6, R6, 0x4
8 ADD  R2, R2, 0x3
```

Listing 4.1: Target part for partial update fault model inference example.

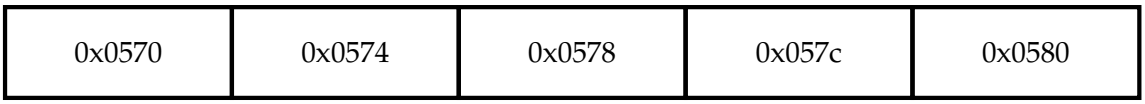
By applying the same analogy to other registers that are not necessarily carrying

```
1 ADD  R1, R1, 0x6
2 ADD  R3, R3, 0xa
3 LSLS R4, R0, 0x4      // 0x0104
4 LSLS R0, R1, 0x10     // 0x0408
5 ADD  R5, R5, 0x1
6 ADD  R2, R6, 0xd
7 ADD  R3, R3, 0x9
8 ADD  R6, R6, 0x4
9 ADD  R2, R2, 0x3
```

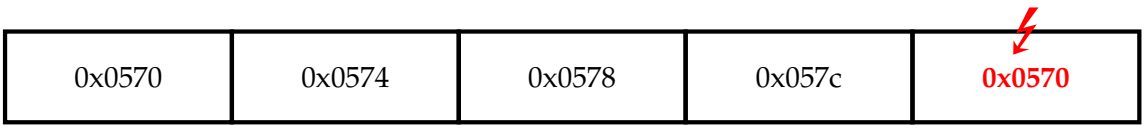
Listing 4.2: Observed exection at ISA when targeting line 3 in Listing 4.1

the binary encoding of the instructions, other faulty behaviors could be explained. For example, Figure 4.7a shows the golden sequence of the memory addresses for the instructions data that are going to be fetched. In contrast, Figure 4.7b shows the fault sequence of the requested addresses that led to have "Non-sequential skip and repeat 32 bits with out-of-order repeat". It is shown that the `fifth` address is partially correctly updated, while part of it kept the same value as before (`0x057-`), which led to repeat the `first` address and skip the `fifth` one. This led to `skip the 5th 32 bits and repeat the 1st 32 bits with out-of-order repeat` at ISA level. Another example, where the address `0x0560` replaced the golden `0x0570`, has also led to same observation.

The next section will provide formal definitions for the sub-cases of the new inferred fault model.



(a) Golden sequence.



(b) Faulty sequence.

Figure 4.7: Non-sequential skip and repeat 32 bits with out-of-order repeat as a result of partial update fault on memory address request data.

4.2.2 Sub-cases of partial update fault model

This section presents the inferred binary encoding fault models that are derived from *partial update* fault model. These fault models seek to explain the faulty behaviors that have been observed after physical fault injection campaigns and do not correspond to one of the previously mentioned fault models in chapter 3. These new fault models can be defined as the following:

Partial update from the precharge value

This fault model corresponds to a fault that happens while the data or fetched instructions are propagated between internal registers from the flash memory to the core, as shown in Figure 4.1.

The hypothesis behind this fault model is based on the fact that not all bits of the data are propagated at the same speed from an internal register to another through a bus or combinational logic. Consequently, not all flip-flops within the destination register will get the update at the same time at a rising edge of a new clock cycle.

In nominal conditions, the clock period is defined such that all signals can be correctly sampled (*i.e.,* the critical path has a positive slack). In case of a clock glitch, however, this behavior is disrupted by the fact the clock edge occurs quite sooner than expected. Thus, with the suitable injection parameters, it may happen that some flip-flops will receive the correct update, while some will receive the precharge value of the bus. Assuming that the precharge value of a bus or a wire between two registers is zero, then the correct update of a flip-flop means receiving the correct logic one or zero, while not receiving the correct update means capturing the precharge value of the bus, *i.e.* zero.

This model is observed as a reset on some bits while the instructions are transferred through the fetch data path in Figure 4.1, as shown experimentally in subsection 4.2.3. Thus, multi bit-reset can be applied to the data to model this fault at RTL.

It should be mentioned that in [72], the authors claimed that some of the observed faults, as a result of electromagnetic fault injection, might be related to the precharge value of the target microcontroller's bus. However, they didn't have a clear model that could explain their observed faults.

Partial update from the previous value

This fault model is somehow similar to the previous one. It occurs on the same path shown in Figure 4.1. However, instead of receiving the precharge value from the bus, some flip-flops within a destination register will keep their old values, either because the values have not been changed or because the corresponding wire still keeps the old values, and hence, the updated values are similar to the old ones. Conversely, and at the same time, other flip-flops in the destination register will receive the correct updated value.

This model is formally described as a bitwise OR between the old value and the new value of an internal register. This merge might be a full merge or a partial merge, as shown experimentally in subsection 4.2.4. Thus, in each flip-flop, the resulting value can be either the previous value or the correct value *i.e.* the value that the flip-flop should receive under normal execution, without any fault injection.

When looking at the execution of the instructions in this case, we observe the following behavior: The instruction(s) fetched at clock cycle i is executed normally. However, the instruction(s) fetched at clock cycle $i+1$ is not the one being executed. Instead, the observed instruction(s) is a full or partial merge between the fetched data at clock cycle i and the fetched data at clock cycle $i+1$. This merge also illustrates how the observed behavior can be simulated at RTL. More details about this behavior are provided in subsection 4.2.4.

Discussion

Exploiting the transition value of a wire or a bus from a precharge (or a previous) value to a new value is a well-established modeling approach in power analysis attacks [125]–[128], which employ the so-called Hamming weight (or distance) leakage model. Likewise, our approach shows a similar pattern: depending on the type of register transition occurring (from previous or precharge value), the corresponding partial update fault model applies.

In subsection 4.2.3 and subsection 4.2.4, it is shown that both cases can occur for the same device. This is not in contrast with our modeling, as depending on the actual element that is affected by the fault injection and the fine-tuning of the injection parameters, we may see different outcomes.

It should also be noticed that these models might not only refer to the binary encod-

ing of the instructions but also to other propagated data, as already shown in subsection 4.2.1.

The results of the different clock glitch fault injection campaigns are discussed and analyzed separately with respect to each fault model in the following subsections.

4.2.3 Experimental results of partial update from precharge value

This section demonstrates how the *partial update from the precharge value* fault model explains many of the faulty behaviors observed during the fault injection campaigns. Also, it demonstrates the relation between this fault model and both the target instruction and the target device. To put it another way, it determines whether some bits in the fetched data are more sensitive to this fault model than other bits and, if so, whether the target instruction or the target device is to blame. These dependency experiments have been performed for the sake of trying to simplify the fault model, and make it more understandable.

The following subsections provide detailed results when targeting different instructions, and also when targeting a new device, identical to the already used one. Additionally, it presents results when targeting a different device with a different brand. The results presented in this section are obtained experimentally when targeting STM32F3 microcontroller. Only in one subsection, as will be indicated, the results refer to an STM32L4.

High-Hamming weight instruction

Since the *partial update from the precharge value* fault model causes some bits of the target instruction to be reset, it makes sense to choose an instruction with a large Hamming Weight in order to maximize the occurrence of the considered fault model. Under this assumption, we chose the instruction `SUBS R6, 0xff`, whose encoding in Thumb-2 is `0x3eff`. Our rationale is twofold: the instruction has a comparatively large Hamming Weight (13) given its size. Secondly, since the *partial update from the precharge value* fault model causes some bits of the target instruction to be reset, applying it on `0x3eff` results in an instruction that is simple to discriminate with high probability.

The objective behind these experiments is to show that several faulty behaviors can be explained using the *partial update from the precharge value* fault model. In addition, we want to see if some bits are more vulnerable than others to be reset within

a target instruction. Finally, were it the case, we need to know if this is because of the target instruction or of the target device. Since the fetch size in the target device is 64 bits, a 16-bit instruction may reside in any of four different positions within these 64 bits. Therefore, four injection campaigns have been performed, one for each possible position of `0x3eff` in the 64-bit word. The main reason of changing the position of the target instruction is to find out if the fault model depends on the target instruction, or it depends on its position within the fetched 64 bits, and hence, depends on the physical implementation of the target device. The remaining three positions are filled with three instructions with the encoding `0x0000`, in order to minimize possible side effects from other instructions and make the analysis easier. This encoding corresponds to the `MOVS R0, R0` instruction, which is equivalent to a `NOP`.

Table 4.2 gives the target part code of each fault injection campaign. It also shows the glitch parameters that are used. These parameters are chosen in order to maximize the number of faults that can be classified under the *partial update from the precharge value* fault model. Position refers to the location of `0x3eff` within the fetched 64 bits. The values of shift and width are provided as a percentage of a single clock cycle: the glitch is introduced before the rising edge of the target clock cycle if shift is negative. ChipWhisperer provides an additional parameter, called fine-width, which is used to offer fine-tuning of the width parameter. It has been noticed that fine-width provides better reproducibility of the results when it is used. Repetitions are the number of executions for each combination of parameters. For each fault injection campaign, the total number of experiments corresponds therefore to more than 20 000 executions, as summarized in the last row of the table. The same value of delay is used in all the campaigns, and it depends on the number of initialization instructions that precede the target part.

The results of the four injection campaigns on `0x3eff` with respect to the three classes (*i.e.,* Crash, Silent and Fault) are presented in Table 4.3. All the resulting faulty behaviors can be classified under two fault models: “Skip” (all the general purpose registers keep their initial values), or *partial update from the precharge value* . Table 4.3 also provides the number of observed behaviors linked to each fault model among the faulty executions.

Figure 4.8 shows the encoding of the **executed** instructions for each injection campaign, along with the number of times each of them is observed. All of these faulty behaviors are classified under the *partial update from the precharge value* fault model.

Position	1st	2nd	3rd	4th
Target part code	`0x3eff` `0x0000` `0x0000` `0x0000`	`0x0000` `0x3eff` `0x0000` `0x0000`	`0x0000` `0x0000` `0x3eff` `0x0000`	`0x0000` `0x0000` `0x0000` `0x3eff`
Shift	-13			
Width	{6, 10}	{6, 10}	{3, 4}	{3, 4}
Fine width	[-255, 255]			
Repetitions	20			
Total	20440			

Table 4.2: Experimental parameters.

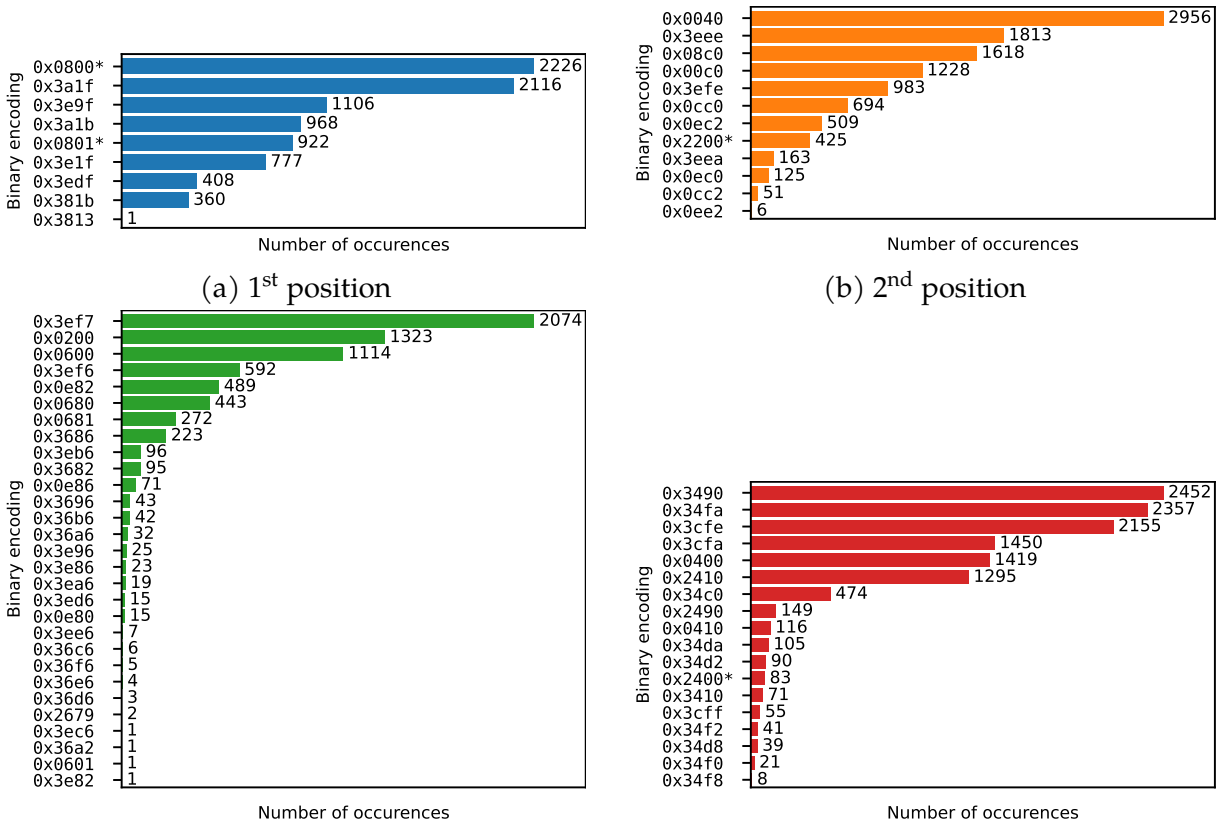
	Position			
Class	1st	2nd	3rd	4th
Crash	0	0	23	1
Silent	33	1574	2273	158
Fault	20 407	18 866	18 144	20 281
Skip	11 523	8295	11 107	7901
Partial update from the precharge value	8884	10 571	7037	12 380

Table 4.3: Fault obtained when targetting the `0x3eff` instruction at four different positions.

This is because all of them can be seen as a reset on some bits of the original instruction `0x3eff`. It should be noticed that resetting all the bits of `0x3eff` will result in executing `0x0000`, which is an instruction with no effect as mentioned earlier, and thus classified under the "Skip" fault model.

Additionally, Table 4.3 and Figure 4.8 show that the number of faulty behaviors and the observed executed instructions depend on the position of the instruction in the fetched 64 bits. Thus, the results depend on the position rather than the target instruction. Furthermore, the results of each position show that some instructions are more probable to be executed than others as a result of the fault injection.

To better understand the effect of the fault at each position, and hence, on each bit in the position, we define a metric called *bit sensitivity*. It measures the probability for a bit to be reset as a result of the clock glitch fault injection over the faulty behaviors



(a) 1st position (b) 2nd position

(c) 3rd position (d) 4th position

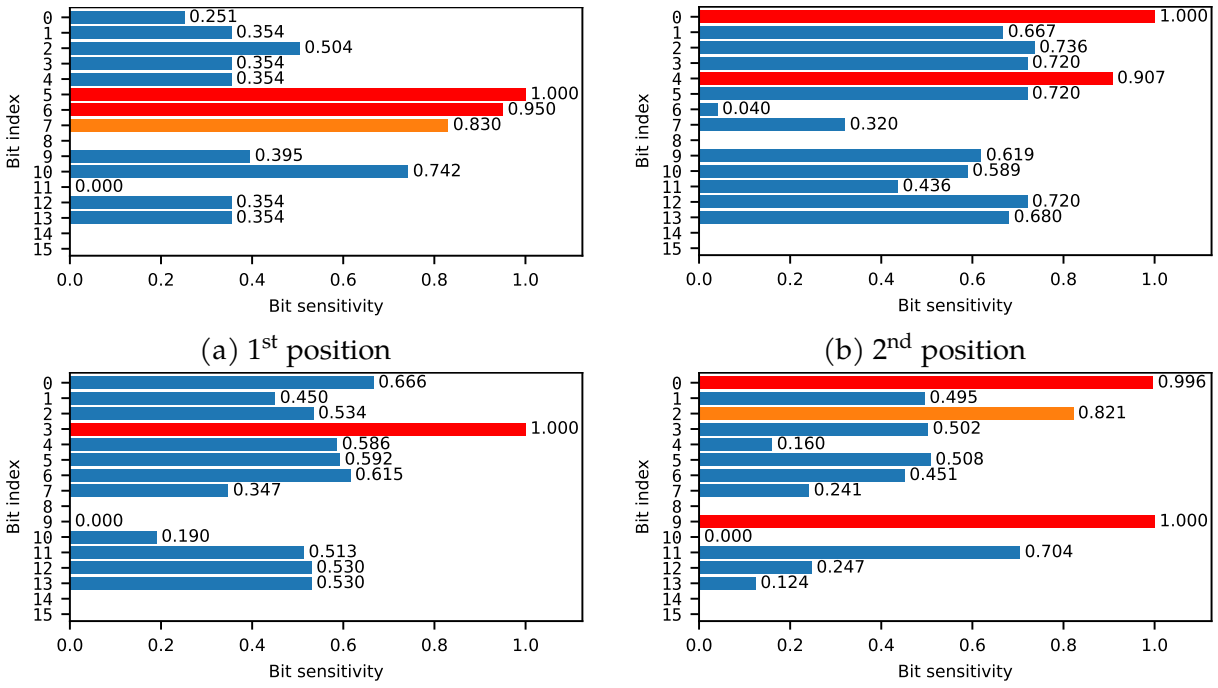
Figure 4.8: Encoding of the observed executed instructions when targeting `0x3eff` at four different positions within the target programs.

that are classified under the *partial update from the precharge value* fault model at a specific position. The *bit sensitivity* $S_p(f, b)$ of bit b to a given fault model f at position p is defined in Equation (4.1).

$$S_p(f, b) = 1 - \frac{P(b = 1 \mid p)}{P(\text{fault model} = f \mid p)} \tag{4.1}$$

Figure 4.9 presents the *bit sensitivity* values for the results obtained during the fault injection campaigns on `0x3eff` at all positions. Obviously, bits that are zero in `0x3eff` (bits 8, 14, and 15) have no corresponding *bit sensitivity* value. We can see that the *bit sensitivity* is different from one position to another and from one bit to another at the same position. Thus, under the *partial update from the precharge value* fault model, some instructions are more probable than others.

Next subsection will present the results of targeting a different instruction, in order



(a) 1st position (b) 2nd position

(c) 3rd position (d) 4th position

Figure 4.9: Bit sensitivity values obtained when targeting `0x3eff` at four different positions.

to confirm that the *partial update from the precharge value* fault model depends on the physical implementation of the device, and not on the target instruction.

It is important to note that whenever there is a doubt about the execution of an instruction, results are confirmed using alternative initial register values. Nonetheless, in rare circumstances, more than one instruction may produce the same outcome, for instance, when the value of a register is zero. For example, this might happen because of moving zero to the register, or by shifting its value by 32 bits. In Figure 4.8, when the encoding is starred, it means that there is an alternative instruction that could lead to the same outcome, and we selected one based on other observed encoding at the same position. It is important to stress that this is happening only in a few cases (4 times), and does not affect the measurements or the general conclusion.

Confirming sensitive bits

Extra experiments have been carried out on `0x3b7d`, which is the encoding of the `SUBS R3, 0x7d` instruction. Again, we chose this instruction since it has a relatively

high Hamming weight, and allows recognizing the encoding of the executed instructions as a result of the *partial update from the precharge value* fault model with high probability. However, we specifically took care to have ones in the most sensitive positions from Figure 4.9 to see if these measurements are reproducible when targeting a different instruction.

The experimental parameters for the fault injection campaigns on `0x3b7d` are identical to that of `0x3eff`, given in Table 4.2. The only difference is that the target program has `0x3b7d` instead of `0x3eff`. The classification results are presented in Table 4.4, while the *bit sensitivity* values are plotted in Figure 4.10.

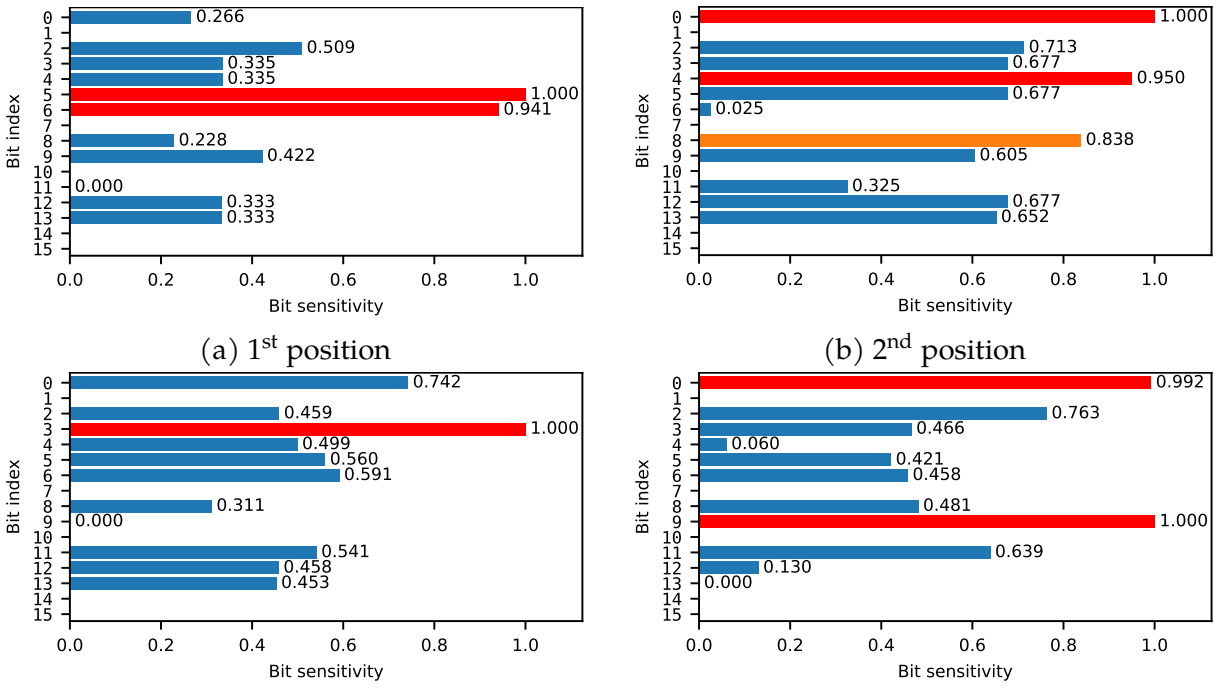
	Position			
Class	1st	2nd	3rd	4th
Crash	0	0	0	0
Silent	39	2304	2589	197
Fault	20 401	18 136	17 851	20 243
Skip	11 694	8386	10 528	7606
Partial update from the precharge value	8707	9750	7323	12 637

Table 4.4: Fault obtained when targeting the `0x37bd` instruction at four different positions.

It is clear that the classification results and the *bit sensitivity* values of `0x3b7d` are very close to the corresponding results of `0x3eff`. This leads us to the conclusion that the *partial update from the precharge value* fault model is instruction-independent. On the other hand, the next subsection shows that *bit sensitivity* greatly depends on the target device.

`0x3eff` experiments on a new STM32F3 microcontroller

In this section, we present the results of targeting the `0x3eff` instruction again while using a brand new device, which we did not use to perform any experiment previously. In any other means, it is identical to the one we used in the previous experiments. This is done to better understand the dependency of the *partial update from the precharge value* fault model on the target device. The experimental parameters of this campaign are identical to those in Table 4.2.



(a) 1st position (b) 2nd position

(c) 3rd position (d) 4th position

Figure 4.10: Bit sensitivity values obtained when targeting `0x3b7d` at four different positions.

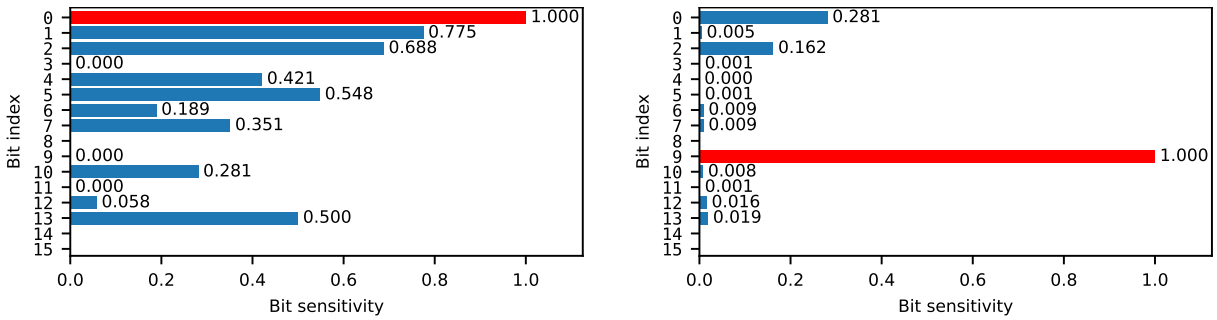
For our purposes, it is enough to present the results on the 2nd and 4th positions to see that they are very different between the two devices. The results are presented in Table 4.5 and Figure 4.11.

A very interesting observation is that the number of faults and the *bit sensitivity* were much higher when performing the fault injection campaigns on the old device. This is clear for the *bit sensitivity* of the 4th position in Figure 4.11b, as we can see the distribution of the bit sensitivities is similar to that in Figures 4.9d and 4.10d, however, on the old device, the sensitivity is much higher. This could be explained as an aging effect, since the old device has been used for fault injection experiments for a few months. We speculate that the *bit sensitivity* could increase over time (a common consequence of performance degradation due to aging), but further research is needed to confirm this observation.

It is also worth mentioning that more observations of the "Skip" fault model may be obtained when targeting the new device. This is achieved by changing the shift value from -13 to -12. Table 4.6 shows the classification results when targeting `0x3eff` while using the new device, with shift = -12. This also gives a clue that only the *partial*

Class	Position 2nd	4th
Crash	0	2
Silent	18 058	16 995
Fault	2382	3443
Skip	345	0
Partial update from the precharge value	2037	3443

Table 4.5: Fault obtained when targeting the `0x3eff` instruction at four different positions on the *new* STM32F3.



(a) 2nd position (b) 4th position

Figure 4.11: Bit sensitivity values obtained when targeting `0x3eff` on the *new* STM32F3 at the 2nd and the 4th positions.

update from the precharge value faults are affected by the fact that this is a new device.

`0x3eff` experiments on an STM32L4 microcontroller

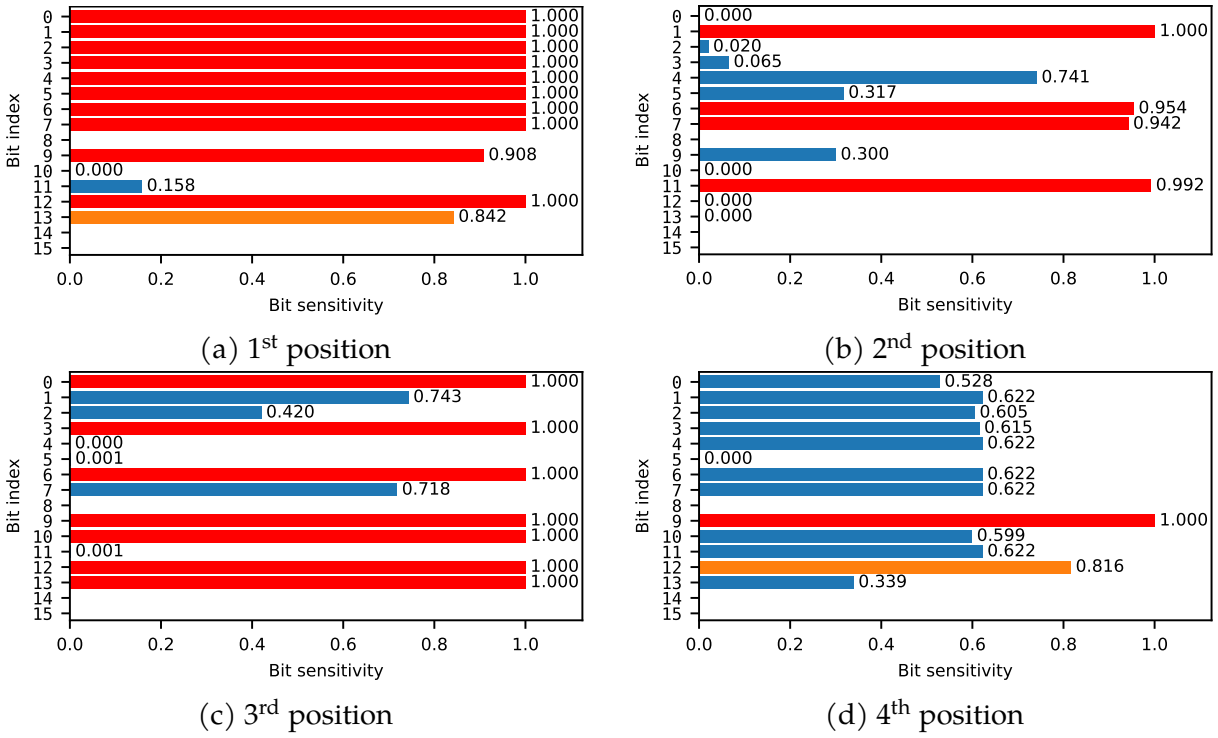
In order to provide more experimental evidences that the *bit sensitivity* is target-dependent, Figure 4.12 shows the *bit sensitivity* values when targeting a different device. It is an STM32L4 microcontroller. It should be mentioned that this target device, has been used for a long time to perform experiments (more than a year). This could explain the high sensitivity in many bit Indices.

Conclusion on bit sensitivity

To summarize, the *bit sensitivity* figures illustrate that, as a result of the *partial update from the precharge value* fault model, the probability distribution of the corrupted

Class	Position 2nd	4th
Crash	178	5
Silent	9205	0
Fault	11 057	20 435
Skip	9526	19 363
Partial update from the precharge value	1531	1072

Table 4.6: Fault obtained when targeting the `0x3eff` instruction on the *new* STM32F3 using a shift value of -12.



(a) 1st position

(b) 2nd position

(c) 3rd position

(d) 4th position

Figure 4.12: Bit sensitivity values obtained when targeting `0x3eff` at four different positions using STM32L4 as a target device.

instruction is not random, and it depends on several features that are mostly device-dependent. The probability of executing a given instruction differs from the probability of executing another. This discrepancy is determined by both the instruction's position inside the target program and the target device itself. This is of prime importance if the instruction results in a security vulnerability, as will be highlighted in chapter 5.

4.2.4 Experimental results of partial update from previous value

In this section, the focus is on the occurrence of faulty behaviors that can be classified under the *partial update from the previous value* fault model. In this case, there is no precharge value and thus the transition occurs from the value that was previously stored in the register. In the time while the register is updating its value, a transient situation may occur when some bits already have the new value, whereas others are still to be updated. We can approximate this behavior as a morphing, or merging, between the previous and the new data.

In the following subsections, we give examples of observed faulty behaviors that can be classified as full or partial merge. It is important to stress that all these examples were experimentally obtained in the campaigns. All the presented examples in this section have been observed experimentally when targeting an STM32F3 microcontroller. Nonetheless, it should be mentioned that similar results (but only in terms of 32 bits) have been observed when targeting an STM32F1 microcontroller. Again as before, the number of bits is related to the flash memory access size. In STM32F3, it is 64 bits, whereas, it is 32 bits in STM32F1.

Full merge

The merge is considered full if and only if the observed executed instruction(s) can be expressed as a bitwise OR between the data fetched at clock cycle i and the data fetched at clock cycle $i+1$. Two examples of this case are provided. The first example shows how *four new* 16-bit instructions are executed as a result of a full merge between eight 16-bit instructions. The second example illustrates how *two new* 32-bit instructions are executed as a result of a full merge between eight different 16-bit instructions.

Executing four new 16-bit instructions Listing 4.3 shows the target program and the binary encoding of each instruction. The green instructions at lines 1 to 4 are fetched at clock cycle i, and the blue instructions at lines 5 to 8 are fetched at clock cycle $i+1$. The bitwise OR between a `MOVS R1, 0x25` instruction and an `ASRS R0, R1, 0x8` instruction results in `ADDS R3, 0x2d`. At the binary encoding level, we have: `0x2125 | 0x1208 = 0x332d`. As a result, the observed execution when performing clock glitch fault injection is given in Listing 4.4. The last four instructions correspond

to four new 16-bit instructions, not present in the initial program.

```
1 MOVS R1, 0x25      // 0x2125
2 MOVS R1, 0x25      // 0x2125
3 MOVS R1, 0x25      // 0x2125
4 MOVS R1, 0x25      // 0x2125
5 ASRS R0, R1, 0x8   // 0x1208
6 ASRS R0, R1, 0x8   // 0x1208
7 ASRS R0, R1, 0x8   // 0x1208
8 ASRS R0, R1, 0x8   // 0x1208
```

Listing 4.3: Target program to execute four new 16-bit instructions as a result of full merge.

```
1 MOVS R1, 0x25   // 0x2125
2 MOVS R1, 0x25   // 0x2125
3 MOVS R1, 0x25   // 0x2125
4 MOVS R1, 0x25   // 0x2125
5 ADDS R3, 0x2d   // 0x332d
6 ADDS R3, 0x2d   // 0x332d
7 ADDS R3, 0x2d   // 0x332d
8 ADDS R3, 0x2d   // 0x332d
```

Listing 4.4: Observed execution as a result of full merge on Listing 4.3.

Executing two new 32-bit instructions Listing 4.5 shows the target program and the encoding of each instruction of this example. The observed execution as a result of the clock glitch fault injection is given in Listing 4.6. The bitwise OR of the first two hexadecimal digits at line 1 (`0xa9`) and the corresponding digits at line 5 (`0x42`) gives `0xeb`. Since the most significant five bits are `0b11101`, this word is decoded as a 32-bit instruction, as explained in chapter 3. The same holds for the merging of instructions at lines 3 and 7.

Listing 4.6 is obtained by a full merge applied on Listing 4.5 in the following way:

- Merging the 32 bits at lines 1 and 2 with the 32 bits at lines 5 and 6 respectively: `0xa9000000 | 0x42000305 = 0xeb000305`.
- Merging the 32 bits at lines 3 and 4 with the 32 bits at lines 7 and 8 respectively: `0xa9000000 | 0x42020405 = 0xeb020405`.

```
1 ADD  R1, SP, 0x0    // 0xa900
2 MOVS R0, R0         // 0x0000
3 ADD  R1, SP, 0x0    // 0xa900
4 MOVS R0, R0         // 0x0000
5 TST  R0, R0         // 0x4200
6 LSLS R5, R0, 0xc    // 0x0305
7 TST  R2, R0         // 0x4202
8 LSLS R5, R0, 0x10   // 0x0405
```

Listing 4.5: Target program to execute two new 32-bit instructions as a result of full merge.

```
1 ADD  R1, SP, 0x0   // 0xa900
2 MOVS R0, R0        // 0x0000
3 ADD  R1, SP, 0x0   // 0xa900
4 MOVS R0, R0        // 0x0000
5 ADD  R3, R0, R5    // 0xeb000305
6 ADD  R4, R2, R5    // 0xeb020405
```

Listing 4.6: Observed execution as a result of full merge on Listing 4.5.

Partial merge

In this case, only part of the fetched data at clock cycle i and the data fetched at clock cycle $i+1$ is merged. Two examples are presented to illustrate this case. The same target code is used for both examples and is shown below in Listing 4.7.

```
1 ADD  R1, R1, 0x4   // 0xf1010104
2 ANDS R2, R0        // 0x4002
3 MOVS R0, R0        // 0x0000
4 ADD  R2, R2, 0xa   // 0xf102020a
5 MOVS R4, R0        // 0x0004
6 MOVS R0, R0        // 0x0000
```

Listing 4.7: Target program of a partial merge.

The first example is a partial merge that occurred over the fetched 64 bits. We observed that not all the 32 bits at lines 1 (`0xf1010104`) are systematically merged with the corresponding 32 bits at line 4 (`0xf102020a`): only the destination and source registers are merged. In addition to this behavior, another partial merge occurred in the following instructions, only over the least significant digit, between the 16 bits at line

2 (0x4002) and the 16 bits at line 5 (0x0004). As a consequence, only the destination register at line 5 is affected. The observed execution of this example is shown in Listing 4.8.

```
1 ADD  R1, R1, 0x4    // 0xf1010104
2 ANDS R2, R0         // 0x4002
3 MOVS R0, R0         // 0x0000
4 ADD  R3, R3, 0xa    // 0xf103030a
5 MOVS R6, R0         // 0x0006
6 MOVS R0, R0         // 0x0000
```

Listing 4.8: Observed execution as a result of partial merge over 64 bits after targeting Listing 4.7.

The second example shows a partial merge where only one instruction is affected, when targeting the same program as in Listing 4.7. The observed execution is presented in Listing 4.9. It is clear that the immediate value (0xa) is updated normally, while the source and the destination registers of both instructions at lines 1 and 4, in Listing 4.7, are merged, so that R3 is affected in the resulting execution in Listing 4.9. In both examples, we cannot discriminate on the opcode values (0xf1), as it is the same in both instructions. It is worth mentioning that a full merge was also observed for the target program in Listing 4.7.

```
1 ADD  R1, R1, 0x4    // 0xf1010104
2 ANDS R2, R0         // 0x4002
3 MOVS R0, R0         // 0x0000
4 ADD  R3, R3, 0xa    // 0xf103030a
5 MOVS R4, R0         // 0x0004
6 MOVS R0, R0         // 0x0000
```

Listing 4.9: Observed execution as a result of partial merge over a single instruction after targeting Listing 4.7.

4.2.5 Conclusion on the results of partial update fault model

Experimental evidences are presented in support of the proposed fault models. We highlighted several different faulty behaviors that support the fault models based on

partial update mechanisms. Depending on the actual low-level implementations, those updates can overwrite previous values or precharge values.

The fault can either affect the full instruction or affect a subset of the instruction encoding. In the former case, the resulting instruction can be easily predicted as the behavior is deterministic; in the latter, the resulting execution is more difficult to predict as the final result depends on the fine-tuning of the injection parameters, and on the device itself as seen in subsection 4.2.3.

It is interesting to note, however, that this behavior may affect instructions that are not adjacent in the memory, as the result of variable-length encoding and fetching microarchitectures. For these reasons, different faulty behaviors, and thus different vulnerabilities, may be obtained for different architectures even if they are supporting the same instruction set. Additionally, simple countermeasures, such as duplication, triplicating, or adding dummy instructions would not be efficient countermeasures, they even may create new vulnerabilities.

4.3 Fault models evaluation

For the RTL fault models, in particular, *preventing the update*, and *anticipating the update*, the Coverage is high as much as the Coverage that was obtained in section 3.6. It reaches, in most cases, more than 90 %. Regarding Fidelity, it is also 100 %. However, it is necessary to note that the RTL fault model takes into account the target register, as not all registers deliver faults similar to the observed faulty behaviors when applying these models to them.

With respect to the *partial update* fault model, the Coverage is increased significantly. For example, in subsection 4.2.3, it is shown that the Coverage is 100 %, as the observed faults are classified either under “Skip” or *partial update from the precharge value* fault models, although the target device is STM32F3, which had the least Coverage in section 3.6 (73.05 %).

Unfortunately, as the analysis of *partial update* fault model is not automated, it is very hard to measure the Coverage for a target program like the ones used in chapter 3. This is because hundreds of faulty behaviors (and sometimes thousands) need to be analyzed in order to classify the observed fault under which model. Also, it could be impossible to do so in some cases, as the combinations, in terms of 64-bit faults, could reach 2^{128} ($2^{64} \times 2^{64}$), which is impossible to automate by simple computers.

Nonetheless, Figure 4.13 shows the Coverage percentage for each fault model when targeting two different devices: STM32F3 and STM32L4. It is obvious that the sum of Coverage for classified faults with known models is more than 99 %. The target program that is used in these campaigns is a series of the same instruction. The instruction is `ADDS R3,0x2b`, which has the encoding of `0x332b`. Targeting a series of the same instruction minimizes the complexity of the analysis. In addition, in this case, we can either observe "Skip", or *partial update from the precharge value* fault models, as effects in terms of "Skip and repeat" or *partial update from the previous value* will be masked, which facilitates the analysis. It is worth mentioning that many faulty behaviors are classified under *partial update from the precharge value* fault model, for instance, in STM32F3 experiments, the 35.51 % belongs to 92 different observations, and the 18.89 % belongs to 36 different faulty behaviors in STM32L4 experiments. In contrast, for "Skip", although the Coverage is higher, it only belongs to a few number of faults. This includes for example, "Skip" four instructions of `0x332b` (*i.e.,* 64 bits), two instructions, or a single instruction.

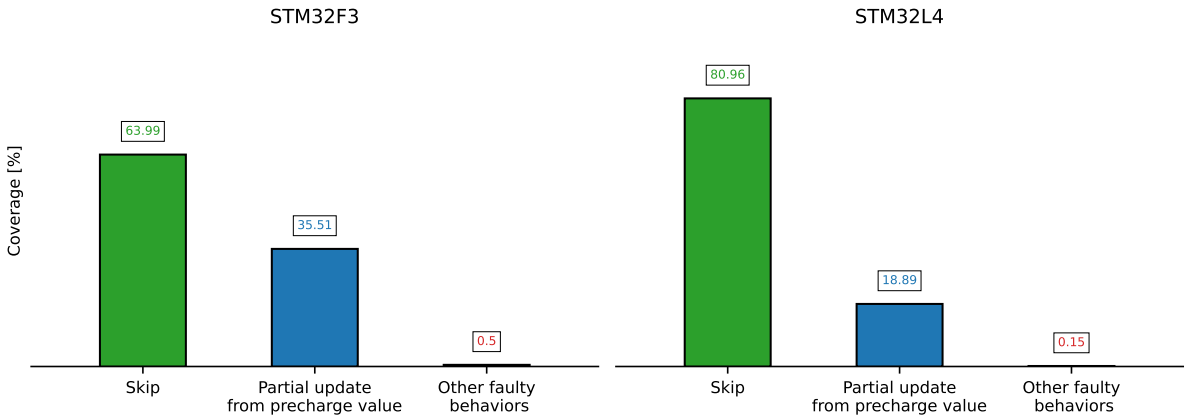


Figure 4.13: Coverage measurement when targeting a series of `0x332b` instruction.

Table 4.7 shows the experimental setup for the aforementioned experiments, along with the classification classes. Small intervals of Width and Shift are used compared to the previous chapter, in order to maximize the Fault %.

Khaut *et. al.* [92] performed laser fault injection campaigns on a series of `0x0000` instruction, in order to see if any fault can be observed other than bit resets. As a result, they did not observe any fault. Inspired by their experiment, a clock glitch fault injection campaign, using the same parameters in Table 4.7, has been conducted on a series

	Target device	
	STM32F3	STM32L4
Shift	[-15,-6]	
Width	[2,11]	
Delay	{39, 40, 41}	
Repetition	100	
Total	30 000	
Silent	73.98 %	76.35 %
Crash	10.07 %	1.66 %
Fault	15.95 %	21.99 %

Table 4.7: Experimental setup and classification cases for fault models evaluation experiments on a series of `0x332b` instruction.

of `0x0000`. As a result, the Fault % in STM32F3 was 0.77 %, and 0.20 % in STM32L4. This is comparable to the Coverage values obtained in Figure 4.13, as the Fault % was 15.95 %, and 21.99 % in STM32F3 and STM32L4 respectively. To put it another way, this experiment shows that the proposed fault models in chapter 3 and chapter 4 have very high Coverage, which is more than 99 %, but not 100 %. This is because targeting a series of `0x0000` would result in a fault that cannot be classified under any of the proposed fault models. Additionally, what makes the proposed models better than the already existing ones in the literature is that the explained faults cannot simply be classified under bit resets as already detailed in previous sections and chapters.

With respect to the Fidelity of *Partial update* fault model, different aspects should be considered: if Fidelity of *Partial update* here means observing any fault that can be classified under *Partial update* fault model, then Fidelity of *Partial update* equals 100 %, and thus, the overall Fidelity also equals 100 %. However, as this fault model is a complicated fault model, the Fidelity here might mean observing all the possible faulty behaviors as a result of applying both *partial update from the precharge value* and *partial update from the previous value* fault models. In this case, the Fidelity of *Partial update* fault model is very low. For example, when targeting single `0x3eff` at `1st` position, Fidelity of *partial update from the precharge value* would equal 100 %, if 2^{13} (8192) executions were observed. However, only 9 different executions were observed. Thus, Fidelity of *partial update from the precharge value* equals 0.1 %, which lowers

noticeably the overall Fidelity. On the other hand, *bit sensitivity* measurement has been defined to show that only a few instructions are possible to be observed as a result of fault injection. Therefore, Fidelity does not necessarily mean observing all the possible executions, but only the most probable ones. Hence, Fidelity is increased significantly. Here it is very hard to quantify the Fidelity, thus, we rather stick to quality. The same Fidelity analysis applies to *partial update from the previous value* in case of Partial merge. However, in the case of Full merge, the Fidelity is 100 %, as it was always observed when targeting STM32F1 and STM32F3.

Regarding Complexity, it is really hard to apply *partial update from the precharge value* and *partial update from the previous value* in the case of Partial merge fault models in vulnerability analysis or countermeasures development. However, having well-defined security proprieties in addition to determining the critical parts of a code, would certainly make this process much easier, and thus, the usage of these models would be clearly beneficial. Section 5.1 will present an example that supports this claim. Also, using other indications, such as the *bit sensitivity*, would help in utilizing these models. On the other hand, applying *partial update from the previous value* in the case of Full merge would be very easy, as this model is very specific and deterministic. Moreover, all the cases of *Partial update* fault model are explainable and understandable at different levels of abstraction, which lowers its complexity. Finally, based on the *bit Sensitivity* measurements, it has been shown that *partial update from the precharge value* fault model is highly device-dependent, and thus, enlarges the complexity. For *partial update from the previous value* in the case of Full merge, it has been observed when targeting STM32F3 and STM32F1, but not STM32L4. Thus, it could also be device-dependent. Nonetheless, the aforementioned experiments showed that all the sub-cases of *Partial update* fault models are program-independent.

4.4 Conclusion

This chapter continued the cross-layer analysis approach by covering more levels of abstraction. In particular, it moved down at system levels and considered the hardware level to perform RTL fault simulation. New RTL fault models have been proposed. These models allowed observing identical faults to the faulty behaviors observed by physical fault injection. Additionally, a fault simulation approach based on critical path analysis has been presented. This approach accelerated the process of the fault sim-

ulation and revealed the origin of the observed faulty behaviors.

Thanks to the RTL analysis, a new fault model at the software level has been inferred: *Partial update* fault model, which is also applicable to RTL level. This model comes in two sub-cases: *partial update from the precharge value* and *partial update from the previous value* . These fault models allow explaining a wide range of the faulty behaviors that are obtained when performing clock glitch fault injection campaigns and were unexplained previously. Therefore, they can be used to perform vulnerability analysis of software codes against fault attacks, and help in better designing efficient and low-cost countermeasures. However, they are more complex than the previously inferred fault models: “Skip” and “Skip and repeat”.

5
Further results and details

While reading this dissertation, several questions may arise. For instance, how can vulnerability analysis be conducted using the proposed models, and how would this aid in proposing cost-effective countermeasures? Moreover, what if multiple glitches are injected during program execution? Additionally, are the proposed fault models still applicable when employing different fault injection techniques? This chapter aims to address these inquiries while also paving the way for new research directions based on this work.

Section 5.1 makes use of the presented fault models to break the control-flow integrity of a program by altering the value of the `Program Counter`, in order to provide an actual application example. Also, it illustrates how understanding the fault effect could lead to propose an efficient countermeasure. Experimental results of injecting multiple glitches are provided in section 5.2. Section 5.3 proves the applicability of the proposed models when performing voltage glitch fault injection experiments. The chapter in concluded in section 5.4.

5.1 Program Counter modification

In this section, the proposed fault models are employed to change the value of the `Program Counter` to an address stored in one of the general-purpose registers. This section uses the exploitation example in section 3.4, however, different scenarios and more details are given here. This illustrates how the fault models can be used to perform vulnerability analysis of a given code, based on a predefined security property.

Additionally, performing fault injection experiments, after this analysis, proves the realism of the proposed fault models. Thus, enabling the design or the development of cost-effective countermeasures.

In the following subsections, the probability of modifying the `Program Counter` has been measured for the target program in Listing 5.1. The results of the different scenarios, along with the fault models that led to the success of the attack, and the glitch parameters that allowed observing the results are summarized in Table 5.1. The success rate is computed over 10 000 executions for each clock glitch fault injection scenario. The glitch parameters are tuned to maximize the success rate. STM32F3 has been used as a target device in these experiments.

```
1  R8 = address of line 11
2  // series of 0x0000
3  ADD R6, R1, 0x4c7   // 0xf20146c7
4  ADD R3, R3, 0xa
5  ADD R4, R4, 0xb
6  ADD R5, R6, R3
7  ADD R3, R3, 0xf
8  // series of 0x0000
9  ADD R5, R5, 0x5
10 // series of 0x0000
11 ADD R1, R1, 0x3
12 ADD R9, R0, R6
```

Listing 5.1: Target program for `Program Counter` modification experiments.

5.1.1 Misaligned code

In section 3.4, we were able to modify the `Program Counter` to an address stored in `R8` as a result of the "Skip" fault model in a misaligned code. This is done by executing the least significant 16 bits of a misaligned 32-bit instruction. The first half of the 32-bit instruction is fetched at clock cycle i and its second half is fetched at clock cycle $i+1$. Therefore, skipping the fetched data at clock cycle i results in decoding the remaining half that is fetched at clock cycle $i+1$, and executing it as a new 16-bit instruction. The same thing can happen for the `ADD R6, R1, 0x4c7` instruction shown in Listing 5.1. Its least significant 16 bits (`0x46c7`) are the encoding of `MOV PC, R8`, which stores the

value of R8 into the Program Counter. Thus, executing MOV PC, R8 leads to a jump from line 3 to line 11, since R8 stores the address of line 11.

The attack is reproduced on Listing 5.1. Many useful and dummy instructions are used in Listing 5.1 to make sure of detecting the execution of MOV PC, R8. The success rate in this scenario was 100 %. It was noticed that 9996 of the executions can be classified under the "Skip" fault model. However, *four* executions can be classified under the *partial update from the precharge value* fault model. This is because resetting some bits of the most significant 16 bits of ADD R6, R1, 0x4c7, will lead to execute two 16-bit instructions, as the most significant five bits do not identify a valid encoding for a 32-bit instruction (as detailed in chapter 3). For these four executions, we confirmed this is happening by observing the values of the registers that the MOVS R1, R0 instruction, of encoding 0x0001, had been executed. Thus, the instructions MOVS R1, R0 and MOV PC, R8 are executed in sequence.

5.1.2 Aligned code

The aforementioned attack relies on the misalignment of the code in memory. we add a single MOVS R0, R0 (0x0000) to the target program, just before ADD R6, R1, 0x4c7 instruction, to realign it. The code is now aligned, all bits of 0xf20146c7 are fetched in a single clock cycle. In this case, the fault model that we can rely on to create new instructions (and thus modify the Program Counter) is the *partial update from the precharge value* fault model. The aim is to reset bits over the most significant 16 bits while not touching the least significant 16 bits, in order to keep the encoding of MOV PC, R8, *i.e.,* 0x46c7. The success rate of the clock glitch fault injection campaign, in this case, was 0.71 %. However, no side effect is observed along with executing MOV PC, R8, but this is normal as resetting some bits of the most significant 16 bits of 0xf20146c7 could lead to execute many 16-bit instructions with no observable effect like MOVS R0, R0 (0x0000), or TST R0, R0 (0x4200) for example.

One could imagine that making the code aligned will protect from the misaligned faulty behaviors that are described in chapter 3. However, this example showed that aligning the code cannot be considered a sufficient countermeasure against clock glitch fault injection attacks, that might focus on misaligned codes. Nonetheless, correct alignment of the sensitive instructions can effectively decrease the success rate of an attack, as demonstrated experimentally.

5.1.3 Countermeasure: register substitution

In this scenario, the code is misaligned, but we changed the destination register in `ADD R6, R1, 0x4c7` from `R6` to `R2`. Other occurrences of `R6` are replaced with `R2` in the rest of the program, as shown in Listing 5.2. Now, the least significant 16-bit word for `ADD R2, R1, 0x4c7` is `0x42c7`. The success rate in this scenario was *zero*: no fault led to modify the `Program Counter` to the value in `R8`, either when using the same experimental parameters that previously achieved a 100 % success rate or with any other parameters.

The `R2` register was chosen because `2` in the instruction encoding can not be turned into a `6` by resetting bits. Thus, we avoid obtaining the encoding of `MOV PC, R8`.

This scenario shows that a clear understanding of the fault effect led to the design of a very simple and cost-effective countermeasure. This proposal clearly has no overhead and can easily be implemented by the compiler, except in rare cases where registers might be under a lot of pressure.

```
1  R8 = address of line 11
2  // series of 0x0000
3  ADD R2, R1, 0x4c7   // 0xf20142c7
4  ADD R3, R3, 0xa
5  ADD R4, R4, 0xb
6  ADD R5, R2, R3
7  ADD R3, R3, 0xf
8  // series of 0x0000
9  ADD R5, R5, 0x5
10 // series of 0x0000
11 ADD R1, R1, 0x3
12 ADD R9, R0, R2
```

Listing 5.2: Protected code against executing `MOV PC, R8`.

5.1.4 Trojan

In this case, dummy code with no effect on the target program is added just before `ADD R2, R1, 0x4c7` in Listing 5.2, where the code is protected against executing `MOV PC, R8`. This dummy code is shown in Listing 5.3. It implements a Trojan that can be activated by clock glitch fault injection in order to controllably exe-

cute the `MOV PC, R8` instruction. It is clear that the *partial update from the previous value* fault model in the full merge case will lead to execute `MOV PC, R8`, since we have that `0x4281 | 0x0446 = 0x46c7 (MOV PC, R8)`. The experimental success rate of this scenario was 95.11 %.

```
1 CMP  R1, R0          // 0x4281
2 MOVS R0, R0          // 0x0000
3 MOVS R0, R0          // 0x0000
4 MOVS R0, R0          // 0x0000
5 LSLS R6, R0, 0x11    // 0x0446
6 MOVS R0, R0          // 0x0000
7 MOVS R0, R0          // 0x0000
8 MOVS R0, R0          // 0x0000
```

Listing 5.3: Dummy code implementing a Trojan.

This scenario is possible if we assume that the attacker is the software developer himself. Alternatively, the compiler used to compile the code may be untrusted and thus represent the attacker in this situation. As a countermeasure, code review or analysis based on the presented fault models would be enough to detect such Trojans.

	Fault injection scenario			
	Misaligned	Aligned	Protected	Trojan
Success rate	100 %	0.71 %	0.0 %	95.11 %
Fault models	Skip (sect. 3.3) (99.96 %) *partial update from the precharge value* (sect. 4.2.3) (0.04 %)	*partial update from the precharge value* (sect. 4.2.3)	-	*partial update from the previous value* (sect. 4.2.4)
Shift	-12	-13	-	-9
Width	3	10	-	4

Table 5.1: Experimental results obtained, and fault injection parameters used when attempting to modify the `Program Counter`.

More on Trojan

It is worth noting that the idea of implementing a Trojan is applicable to any of the proposed fault models. Therefore, it also validates the reliability as well as trustworthiness of the presented fault models.

As another example, Listing 5.4 shows a simple password verification code of five characters. `R4` is used as the loop counter. The instruction `ADDw R6, R6, 0xc` represents the dummy code that implements the Trojan. It has the encoding of `0xf106060c`. The least significant 16 bits (`0x060c`) is a valid encoding of the 16-bit instruction `LSLS R4, R1, 0x18`. The initial value of `R1` was 9. Thus, executing `LSLS R4, R1, 0x18` would corrupt the value of `R4` to be 150 994 944 $(= 9 << 24)$, which is highly greater than 4. As a result, the loop condition is violated, and then, the checked password is authenticated even if it is a wrong password. Executing `LSLS R4, R1, 0x18` can be obtained as a result of applying "Skip" or *partial update from the precharge value* fault models, as already detailed. `NOPs` are added to make sure that the injection does not affect the original loop and to ensure that the code is misaligned. The success rate of this example was 100 % after 100 executions: 97 of them are classified under "Skip" fault model, while the remaining 3 executions are classified under *partial update from the precharge value* fault model.

```
1  //pw: the sent text to be checked
2  //passwd: the correct password
3  passok = 1;
4  R4 = 0;
5  // start of inline assembly code
6  // NOPs
7  ADDw R6, R6, 0xc // 0xf106060c
8  // NOPs
9  // end of inline assembly code
10 while(R4 < 5){
11     if(pw[R4] != passwd[R4]){
12         passok=0;
13     }
14     R4++;
15 }
16 return passok;
```

Listing 5.4: Bypassing password verification using a Trojan.

Applying the same idea on advanced and protected programs would be a very interesting perspective of this work.

5.2 Multiple glitch fault injection

Recent works started paying attention to the ability of an attacker to inject multi-faults at different times [78], [129], [130]. Studying and characterizing this type of fault injection would be necessary in two-fold: firstly, to see if an attacker would be able to combine faulty behaviors, where each of these faulty behaviors can be achieved while injecting a single fault. Thus, the need to apply combinations of the proposed fault models while analyzing code vulnerabilities. Secondly, to see if injecting multi-faults would lead to create new faulty behaviors that are not simply a combination of single faults. Thus, the need to apply again the proposed methodology to infer new fault models. Nevertheless, we need to keep in mind that injection multiple faults in real life scenarios is mush more difficult than injecting a single fault.

This section shows the ability to combine different faulty behaviors as a result of injecting multiple glitches. On the other hand, investigating the ability to observe totally new faulty behaviors will be an important future work to look at.

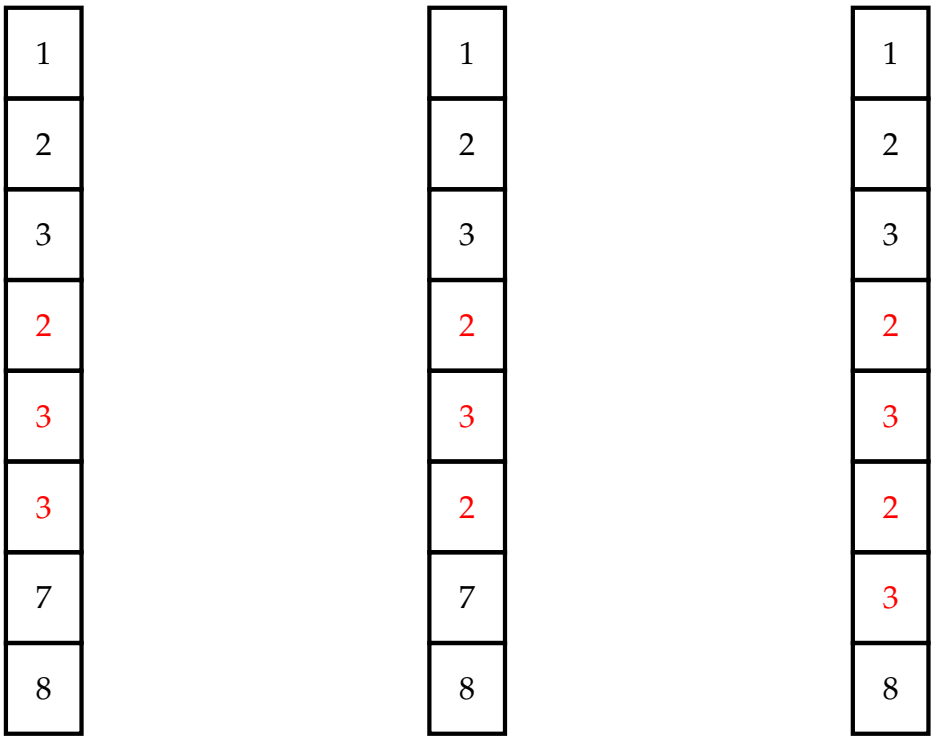
ChipWhisperer environment allows injecting multiple consecutive glitches at a given delay value. This capability is leveraged to perform multiple glitch fault injections. STM32L4 was used in the experiments of this section.

As an example, two glitches have been injected while executing the target code in Listing 5.5. As a result, effects that can be explained as a combination of the proposed fault models have been observed. Figure 5.1 shows three different observations as a result of combining faults under “Skip and repeat” fault model. It should be noticed that targeting the instructions at lines 2 and 3 in Listing 5.5 allows detecting the orders of execution that are obtained in Figure 5.1. Other operands have also been used to confirm these orders of execution.

```
1 46a00602 // MOV R8, R4 // LSLS R2, R0, 0x18
2 f2030109 // ADDw R1, R3, 0x9
3 f2010308 // ADDw R3, R1, 0x8
4 f104040b // ADD R4, R4, 0xb
5 eb060503 // ADD R5, R6, R3
6 f103030f // ADD R3, R3, 0xf
7 f1060607 // ADD R6, R6, 0x7
8 f1050505 // ADD R5, R5, 0x5
```

Listing 5.5: Binary encoding of the aligned target code, for multiple glitch experiments, in hexadecimal format.

Figure 5.1a represents a combination of “Skip and repeat 64 bits” and “Skip and repeat 32 bits”, while Figure 5.1b illustrates a combination of “Skip and repeat 64 bits” and “non-sequential Skip and repeat 32 bits”. On the other hand, Figure 5.1c shows a combination of two “Skip and repeat 64 bits”.



(a) Skip 4, 5, 6 & repeat 2, 3, 3. (b) Skip 4, 5, 6 & repeat 2, 3, 2. (c) Skip 4, 5, 6 ,7 & repeat 2, 3, 2, 3.

Figure 5.1: Observed executions of combined “Skip and repeat” faults when targeting Listing 5.5 as a result of injecting two glitches. Numbers in this figure refer to the encoding at the line numbers in Listing 5.5.

Similarly, targeting the misaligned code in Listing 5.6 also allowed observing a combination of faulty behaviors, where each of them can be explained as a result of a single fault that is affecting a misaligned code. This code is the same as the misaligned code used in subsection 3.3.2. Table 5.2 shows the number of injected glitches that allowed observing the corresponding faulty behaviors in the second column when targeting Listing 5.6. These faulty behaviors are explained as a combination of “Skip 32 bits” faults starting from the first line in Listing 5.6.

These examples show that different codes or alignments will not have an effect on the ability to combine faulty behaviors as a result of injecting multiple glitches. It should also be noticed that the effect of each injected glitch does not necessarily correspond to

```
1 0402f101
2 0106f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00
```

Listing 5.6: Binary encoding of the misaligned target code, for multiple glitch experiments, in hexadecimal format.

an effective glitch that causes a faulty behavior. For example, sometimes even though more than one glitch is injected, the effect is similar to injecting only one glitch. This is obvious in Table 5.2, for instance, “Skip” line 1 is observed when injecting either 1, 2, or 3 glitches.

Number of glitches	Faulty behavior
1, 2, or 3	Skip line 1
2 or 3	Skip lines 1 and 2
3	Skip lines 1, 2, and 3

Table 5.2: Combination of Skip faults when targeting Listing 5.6, using multiple glitches.

5.3 Voltage glitch fault injection

ChipWhisperer supports performing voltage glitch fault injection with the same parameters as clock glitch. In which, Shift determines the offset of the glitch with respect to the targeted clock cycle. Width represents the Length of the voltage glitch. While the pulse amplitude is automatically configured with a voltage drop to zero using a switch MOSFET transistor, which shorts the power line to ground (GND), instead of the normal voltage supply.

“Skip”, “Skip and repeat”, “non-sequential skip and repeat”, *partial update from the precharge value* , and *partial update from the previous value* fault models have also been observed when performing voltage glitch fault injection experiments. Nonetheless, it was noticed that the Coverage of “Skip” and “Skip and repeat” fault models,

using voltage glitch, is less than their Coverage when performing clock glitch. In contrast, faults under *Partial update fault model*, using voltage glitch, were more than the corresponding ones using clock glitch, as will be detailed in the following.

Table 5.3 illustrates the results in terms of Silent, Crash, and Fault when targeting Listing 5.7 in large voltage glitch fault injection campaigns. This target program is the same program used for fault models evaluation in section 3.6. Table 5.3 also shows the Coverage percentage within the observed faulty behaviors. This Coverage includes the faults that are classified under "Skip", "Skip and repeat", and "non-sequential skip and repeat", but not under *Partial update* fault model. It is shown that these values are less than the corresponding ones using clock glitch: using clock glitch for STM32F3, the Coverage was 73.05 %, and for STM32L4, the Coverage was 93.28 %, as shown in Table 5.3.

```
1 ADD  R1, R1, 0x6
2 ADD  R3, R3, 0xa
3 ADD  R4, R4, 0xb
4 ADD  R5, R5, 0x1
5 ADD  R2, R6, 0xd
6 ADD  R3, R3, 0x9
7 ADD  R6, R6, 0x4
8 ADD  R2, R2, 0x3
```

Listing 5.7: Target part for fault models evaluation campaigns using voltage glitch.

	Target device			
	STM32F3		**STM32L4**	
	voltage	clock	voltage	clock
Silent	84.16	96.84	95.45	96.4
Crash	14.19	1.95	3.53	0.36
Fault	1.65	1.21	1.02	3.24
Coverage	56.57	73.05	84.44	93.28

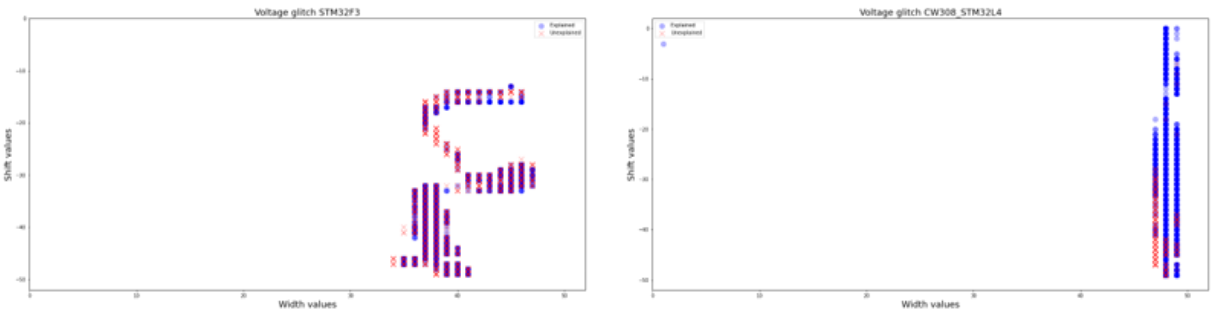
Table 5.3: Experimental results for fault models evaluation experiments using voltage and clock glitch when targeting Listing 5.7. Values in %. Coverage does not include faults under *Partial update* fault model.

Table 5.4 shows the parameters used in these voltage glitch campaigns. The num-

ber of injected glitches is 3 for STM32L4, as performing a voltage glitch using CW308 targets requires more than one voltage glitch to have a single effective glitch. This might be related to the fact that the glitch goes through a long SMA wire, which lowers its effectiveness. Thus, more consecutive glitches increase the overall length of the glitch, and hence, the more probable the success of the glitch will be. Another observation is that specific regions of Shift and Width allowed observing faulty behaviors with either explained or unexplained faults, as shown in Figure 5.2. Again, this figure does not consider faults under *Partial update* fault model. Blue circles ● mean the occurred fault is classified under either "Skip", "Skip and repeat", or "non-sequential skip and repeat", while the red x corresponds to other faults.

	Target device	
	STM32F3	STM32L4
Shift	[-49,0]	
Width	[0,49]	
Delay	{38, 39, 40}	{35, 36, 37}
# of glitches	1	3
Repetition	100	
Total	750 000	

Table 5.4: Experimental setup for fault models evaluation experiments, using voltage glitch, when targeting Listing 5.7.



(a) STM32F3 results (b) STM32L4 results

Figure 5.2: Faults classification with respect to shift and width values using voltage glitch campaigns. Y-axis: Shift ∈ [-49,0], X-axis: Width ∈ [0,49].

On the other hand, Figure 5.3 shows the Coverage values of "Skip" and *partial up-*

date from the precharge value fault models, when carrying out voltage glitch campaigns on a series of `0x332b` instruction. These campaigns are similar to the experiments performed in section 4.3. The overall Coverage is very high in both devices: 92.55 % in STM32F3, and 100 % in STM32L4. Also, it is clear that faults under *partial update from the precharge value* are more than faults under "Skip" when targeting STM32F3. This explains why low Coverage values were obtained when considering only fault models other than *Partial update* fault model. Table 5.5 shows the experimental setup parameters that are used and the classification cases that are obtained when targeting series of `0x332b` instruction using voltage glitch fault injection.

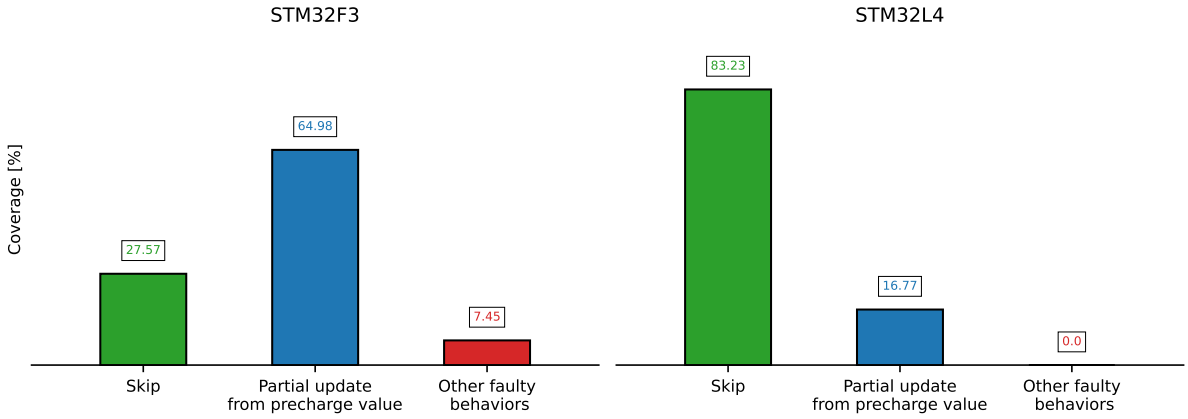


Figure 5.3: Coverage measurement when targeting a series of `0x332b` instruction using voltage glitch.

Finally, as an example of *partial update from the previous value* fault model in the case of Full merge; performing voltage glitch using STM32F3, on the Trojan code in Listing 5.3 in subsection 5.1.4 led to execute `MOV PC, R8` with success rate 92.10 % (using width = 40, shift = −17). This is comparable to the success rate obtained using clock glitch (95.11 %).

5.4 Conclusion

In this chapter, additional analysis, results, and details are provided. Also, new research directions can be followed based on the work presented in this chapter. Firstly, different scenarios to break the control-flow integrity of a program are presented. These scenarios demonstrated how a security evaluator can predict a faulty behavior based

	Target device	
	STM32F3	STM32L4
Shift	[-35,-26]	
Width	{47, 48, 49}	
Delay	{35, 36, 37}	
Repetition	350	
Total	31 500	
Silent	67.5 %	68.69 %
Crash	26.67 %	4.13 %
Fault	5.83 %	27.18 %

Table 5.5: Experimental setup and classification cases for fault models evaluation experiments on a series of `0x332b` instruction using voltage glitch.

on the proposed fault models. Moreover, experimental results proved the realism of this prediction, and thus, proved the reliability of the proposed fault models. Furthermore, understanding the possible effect of the fault allowed protecting the code with simple countermeasures that have no additional overhead. In addition to that, Trojan idea was presented, which also increased the trustworthiness of the proposed fault models. In addition, it shows a new Threat model as a result of fault injection. In which, the software developer or the used compiler represents the adversary.

Moreover, this chapter shows the applicability of injecting multiple glitches to combine faulty behaviors, where each of them can be obtained as a result of a single glitch. This increases the abilities of an attacker to obtain more exploitable faults. At the same time, it raises the complexity from a protection point of view. Looking for new faulty effects as a result of injecting multiple faults would be an interesting perspective of this work.

Finally, performing voltage glitch fault injection campaigns validated the applicability of the proposed fault models when conducting different fault injection techniques from clock glitch. It was shown that in some campaigns, the Coverage reached 100 %. This confirms the high Fidelity and soundness of the proposed fault models even when a distinct injection technique is employed. The possibility of witnessing faulty behaviors using voltage glitch that are distinct from faulty behaviors observed with clock glitch may be explored in a future study.

CONCLUSION AND PERSPECTIVES

As digital systems become more prevalent and the IoT market experiences significant growth, there has been an increased focus on security among hardware designers and software developers. The objective is to protect these devices from potential threats. Among the various security concerns, physical attacks pose a significant risk, with fault injection being a major form of such attacks.

In order to assess digital systems against fault injection attacks and protect them from such threats, software developers and hardware designers must rely on realistic fault models. However, the complexity of embedded microprocessors and their behavior under these attacks pose considerable challenges when attempting to derive accurate fault models based on limited observations of faulty microprocessors, especially when the analysis is limited to a single level of abstraction. Moreover, relying on impractical or random fault models would lead to inadequate and inaccurate vulnerability analysis, potentially resulting in the development or design of countermeasures that are either over or under-engineered.

The aim of this thesis was to provide a cross-layer analysis approach to better analyze, characterize, and understand the effects of fault injection attacks at different levels of abstraction of a digital system. Therefore, trustworthy and reliable software and hardware fault models should be offered, enabling thorough vulnerability analysis processes as well as effective design and/or development of countermeasures.

Summary of the contributions

In this thesis, fault injection techniques that affect the timing constraints in digital systems, along with examples of real attacks from the literature, have been presented in chapter 1. In addition, chapter 1 examined various state-of-the-art studies dealing with the characterization and modeling of fault effects. These studies, combined with the experimental evidence presented in chapter 2, highlighted the necessity of bridging the gap between previous research efforts in order to conduct a more comprehensive analysis and gain a deeper understanding of the implications of the fault injection. From

there, chapter 2 introduced a cross-layer analysis methodology aimed at optimizing and inferring fault models for both software and hardware levels of abstraction. This methodology involves conducting physical fault injection campaigns, following the propagation of faults, performing multiple series of hardware fault simulation experiments, and describing and simulating the fault effects across various software levels. Most importantly is that this proposed methodology does not depend on the target devices, the target programs, and the fault injection techniques.

By implementing the proposed methodology, it became possible to derive fault models that are realistic and explainable. Chapter 3 illustrated the inference process for the two main fault models: “Skip” and “Skip and repeat” for a specific number of bits, which are applied to the binary encoding of the instructions. The realism of these fault models has been substantiated by their ability to anticipate potential faulty behaviors even before conducting actual fault injection experiments, demonstrating their high Fidelity. Furthermore, these two fault models, along with the “non-sequential skip and repeat” model, effectively account for the majority of observed faulty behaviors across different target programs and devices. In most cases, when performing clock glitch fault injection campaigns on different STM32 devices, the Coverage of these models reached approximately 90 %. Chapter 3 also included an experimental proof-of-concept exploitation example and vulnerability analysis of three distinct implementations of the AES block cipher based on the proposed fault models, showcasing their practical applicability at the application level.

Proceeding to the hardware level within the analysis process, as depicted in chapter 4, facilitated the inference of two new RTL fault models. These fault models, namely *anticipating the update*, and *preventing the update*, led to observe RTL faults that are identical to real faults and those generated by higher-level fault models, *i.e.,* “Skip” and “Skip and repeat” fault models. Moreover, understanding the propagation of faults at the hardware level resulted in the inference of another novel fault model: *Partial update* fault model, which is applicable to both levels, RTL and binary encoding of instructions. The *Partial update* fault model encompasses two variations: the *partial update from the precharge value* and the *partial update from the previous value* fault models. The *Partial update* fault model has significantly enhanced the Coverage of the faulty behaviors, reaching, in some cases, up to 100 %. Nevertheless, it has been shown that the *Partial update* fault model is significantly more complex than the others. This is because the *Partial update* fault model is more generic than the others. Additionally, it is

highly device-dependent, making it more challenging to accurately predict the potential faulty behaviors that may arise from this model.

Finally, chapter 5 opens the doors to new research directions based on the presented work. Firstly, the proposed fault models are exploited to intentionally modify the value of the `Program Counter`, thereby compromising the control-flow integrity of the target program. This demonstration showcased how the proposed models can contribute to the vulnerability analysis process. Consequently, a simple and cost-effective countermeasure was proposed, referred to as *register substitution*. Additionally, an experimental presentation of a novel threat model, focusing on Trojan activation, provided further confirmation of the high Fidelity of the proposed fault models. Furthermore, chapter 5 showed the capability to combine diverse faulty behaviors resulting from the injection of multiple glitches. Lastly, the suitability of the fault models for fault injection using a different technique than clock glitch has been validated through the conduction of voltage glitch fault injection. This demonstrated the versatility and effectiveness of the fault models across different fault injection techniques. Notably, for voltage glitch experiments on the STM32L4 target, the Coverage reached 100 % exactly.

Perspectives

In addition to what has already been presented in chapter 5, various research directions could be explored based on this dissertation. First of all, it would be crucial to target devices that embed processors different from the Arm Cortex-M processors. Particularly, RISC-V-based architectures are an intriguing target due to the growing interest towards such architectures. This is mainly due to the availability of open-source hardware descriptions for RISC-V processors, which would greatly facilitate performing cross-layer analysis on these processors. Additionally, conducting fault injection attacks on more complex processors like Intel and AMD, especially considering their support for variable-length instruction sets, would be highly interesting. Validating the applicability of the provided fault models on different target processors would not only lower their complexity but also facilitate a unified vulnerability analysis process, and thus, enable the development of hardware-independent countermeasures. Nevertheless, it is possible that some faulty behaviors may arise that cannot be categorized under the proposed fault models. In such cases, applying the same methodology presented in this work would undoubtedly help in proposing new fault models capable of

explaining these new faulty behaviors.

Similarly, carrying out fault injection using more advanced techniques, such as EM or laser fault injection, presents another interesting perspective. This would provide validation for the effectiveness of the inferred fault models when employing advanced and local fault injection methods. However, it should be noted that different injection techniques may yield diverse faulty behaviors that impact various stages of a processor's pipeline. In such cases, applying the proposed methodology would certainly help in understanding these new effects. Nonetheless, it is important to note that if a different fault injection technique is utilized, especially a one that does not result in timing violations, it becomes necessary to employ an RTL fault simulation method that relies on a technique distinct from path delay analysis. This may encompass a divide and conquer approach for the microarchitectural components, followed by random or exhaustive bit-manipulation fault simulation. Furthermore, investigating the implementation of a clock or voltage glitch in a software-based manner, to manipulate energy management systems for example, represents another interesting perspective for future exploration. This would allow for the application of fault injection in different ways and techniques.

Additionally, proper formalization of countermeasures against the presented fault models, while keeping the best performance possible, will be very important and necessary. These countermeasures could be investigated both at software and hardware levels. At the software level, an automated framework made of vulnerability assessment followed by automatic code protection would greatly improve the security of the target application. This can particularly be integrated at the compiler level for example. At a lower level, several approaches might be envisioned at different abstraction levels, from ISA down to transistor-level approaches.

Moreover, conducting an evaluation of real-life security applications, beyond AES, utilizing the proposed fault models would be another interesting avenue for future research. This exploration would demonstrate the effectiveness of the fault models in detecting software vulnerabilities that could be exploited through fault injection attacks. Integrating deep learning techniques into vulnerability analysis for such scenarios would be particularly captivating, especially when assessing large-scale applications comprising thousands of lines of code. Based on predefined security properties, such AI techniques could help in finding critical parts within a code. These parts may have an effect on the security properties. Thus, the use such AI techniques would make the

process of vulnerability analysis easier.

Finally, another research direction that could be considered is a deep investigation of the effects of aging and the environmental conditions on the success rate of the fault injection attacks, especially with respect to the *Partial update* fault model.

PUBLICATIONS AND COMMUNICATIONS

Peer-reviewed journals

- **Published:** I. Alshaer, B. Colombier, C. Deleuze, P. Maistri and V. Beroulle, “Cross-layer inference methodology for microarchitecture-aware fault models”, *Microelectronics Reliability*, Volume 139, 2022, 114841, ISSN 0026-2714, https://doi.org/10.1016/j.microrel.2022.114841.

- **Submitted:** I. Alshaer, G. Burghoorn, B. Colombier, C. Deleuze, V. Beroulle, and P. Maistri “Cross-Layer Analysis of Clock Glitch Fault Injection on a Variable-length Instruction Set”. This paper concerns sections: 3.2, 3.3, 3.4, and 4.1.

International peer-reviewed conferences with proceedings

- **Published:** I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle and P. Maistri, “Microarchitecture-aware Fault Models: Experimental Evidence and Cross-Layer Inference Methodology”, *2021 16th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, Virual event, Apulia, Italy, 2021, pp. 1-6, doi: 10.1109/DTIS53253.2021.9505074.

- **Published, Best paper award:** I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle and P. Maistri, “Variable-Length Instruction Set: Feature or Bug?,” *2022 25th Euromicro Conference on Digital System Design (DSD)*, Maspalomas, Spain, 2022, pp. 464-471, doi: 10.1109/DSD57027.2022.00068.

- **Accepted:** I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle and P. Maistri, “Microarchitectural Insights into Unexplained Behaviors under Clock Glitch Fault Injection”. *Smart Card Research and Advanced Applications - 22th International Conference, CARDIS 2023, Amsterdam, Netherlands, November, 2023*

Talks

- I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle and P. Maistri,“Variable-Length Instruction Set: Feature or Bug?,” *2022 Journée thématique sur les attaques par injection de fautes (JAIF)*, Valence, France, November 2022.
- I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle, P. Maistri, The Right Level for Fault Modelling, Virual event, Hardware Security Symposium, Amrita University, Kerala, India, January, 2023.
- I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle and P. Maistri, “Variable-Length Instruction Set: Feature or Bug?,” *Forum International De La Cybersécurité (FIC)*, Hacking Lab, Lille, France, April 2023.

Posters

- I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle and P. Maistri,“Cross-Layer Fault Analysis for Microprocessor Architectures,” *2022 Journée thématique sur les attaques par injection de fautes (JAIF)*, Valence, France, November 2022.
- I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle and P. Maistri,“Cross-Layer Fault Analysis for Microprocessor Architectures,” *2022 International Winter School on Microarchitectural security*, Paris, France, December 2022.
- I. Alshaer, S. Michelland, A. Al-Kaf, V. Beroulle, C. Deleuze, , V. Egloff, L. Gonnord and D. Hely, “From hardware vulnerabilities to combined hardware/software countermeasure integration,” *2023 Journée thématique sur les attaques par injection de fautes (JAIF)*, Gardanne, France, September 2023.

BIBLIOGRAPHY

[1] Lionel Sujay Vailshery, *Number of IoT connected devices worldwide 2019-2021, with forecasts to 2030.* [Accessed: July 3, 2023]. [Online]. Available: `https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/`.

[2] W. Wang, Y. Yu, F. Standaert, J. Liu, Z. Guo, and D. Gu, "Ridge-based DPA: improvement of differential power analysis for nanoscale chips", *IEEE Trans. Inf. Forensics Secur.*, vol. 13, *5*, pp. 1301–1316, 2018.

[3] A. Sayakkara, N.-A. Le-Khac, and M. Scanlon, "A survey of electromagnetic side-channel attacks and discussion on their case-progressing potential for digital forensics", *Digital Investigation*, vol. 29, pp. 43–54, 2019, ISSN: 1742-2876.

[4] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR", *in 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, IEEE Computer Society, 2013, pp. 191–205.

[5] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of eliminating errors in cryptographic computations", *J. Cryptology*, vol. 14, pp. 101–119, 2001.

[6] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies", *IEEE Transactions on Device and Materials Reliability*, vol. 5, *3*, pp. 305–316, 2005.

[7] B. Colombier, A. Menu, J.-M. Dutertre, P.-A. Moëllic, J.-B. Rigaud, and J.-L. Danger, "Laser-induced single-bit faults in flash memory: instructions corruption on a 32-bit microcontroller", *in 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019, pp. 1–10.

[8] J. Proy, K. Heydemann, A. Berzati, F. Majéric, and A. Cohen, "A first ISA-level characterization of EM pulse effects on superscalar microarchitectures: A secure software perspective", *in Proceedings of the 14th International Conference on Availability, Reliability and Security, ARES 2019, Canterbury, UK, August 26-29, 2019*, ACM, 2019, 7:1–7:10.

[9] N. Timmers and C. Mune, "Escalating privileges in linux using voltage fault injection", *in 2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017*, IEEE Computer Society, 2017, pp. 1–8.

[10] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont, "Software fault resistance is futile: effective single-glitch attacks", *in 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2016, pp. 47–58.

[11] S. Skorobogatov, "Local heating attacks on flash memory devices", *in 2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, 2009, pp. 1–6.

[12] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "Voltjockey: breaching trustzone by software-controlled voltage manipulation over multi-core frequencies", *in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds., ACM, 2019, pp. 195–209.

[13] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A. Sadeghi, "V0ltpwn: attacking x86 processor integrity from software", *in 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds., USENIX Association, 2020, pp. 1445–1461.

[14] K. Murdock, D. F. Oswald, F. D. Garcia, J. V. Bulck, F. Piessens, and D. Gruss, "Plundervolt: how a little bit of undervolting can create a lot of trouble", *IEEE Secur. Priv.*, vol. 18, *5*, pp. 28–37, 2020.

[15] S. L. Harris and D. M. Harris, "3 - sequential logic design", *in Digital Design and Computer Architecture*, S. L. Harris and D. M. Harris, Eds., Boston: Morgan Kaufmann, 2016, pp. 108–171, ISBN: 978-0-12-800056-4.

[16] Ankit Mahajan, *Relation between clock skew and frequency of operation.* [Accessed: July 3, 2022]. [Online]. Available: `https://www.linkedin.com/pulse/relation-between-skew-frequency-operation-ankit-mahajan/`.

[17] Texas Instruments, *Basics of spi: timing requirements and switching characteristics.* https://training.ti.com/sites/default/files/docs/adcs-spi-communications-timing-presentation.pdf, [Accessed: July 3, 2022].

[18] D. Markovic, B. Nikolic, and R. Brodersen, "Analysis and design of low-energy flip-flops", *in Proceedings of the 2001 international symposium on Low power electronics and design*, 2001, pp. 52–55.

[19] J. Breier and X. Hou, "How practical are fault injection attacks, really?", *IEEE Access*, vol. 10, pp. 113 122–113 130, 2022.

[20] J. Schmidt and C. Herbst, "A practical fault attack on square and multiply", *in Fifth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2008, FDTC 2008, Washington, DC, USA, 10 August 2008*, L. Breveglieri, S. Gueron, I. Koren, D. Naccache, and J. Seifert, Eds., IEEE Computer Society, 2008, pp. 53–58.

[21] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Commun. ACM*, vol. 21, *2*, pp. 120–126, Feb. 1978, ISSN: 0001-0782.

[22] J. Guo, T. Peyrin, A. Poschmann, and M. J. B. Robshaw, "The LED block cipher", *IACR Cryptol. ePrint Arch.*, p. 600, 2012.

[23] Gaisler Research, *Leon3 processor*, [Accessed: March 22, 2023]. [Online]. Available: `https://www.gaisler.com/index.php/products/processors/leon3`.

[24] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems", *in Advances in Cryptology — CRYPTO '97*, B. S. Kaliski, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 513–525.

[25] C. Dobraunig, M. Eichlseder, T. Korak, S. Mangard, F. Mendel, and R. Primas, "SIFA: exploiting ineffective fault inductions on symmetric cryptography", *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, *3*, pp. 547–572, 2018.

[26] J. Daemen and V. Rijmen, "Rijndael for AES", *in The Third Advanced Encryption Standard Candidate Conference*, New York, USA: National Institute of Standards and Technology, Apr. 2000, pp. 343–348.

[27] G. Bai, S. Bobba, and I. Hjj, "Static timing analysis including power supply noise effect on propagation delay in vlsi circuits", *in Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 2001, pp. 295–300.

[28] J. Chen, H. Kando, T. Kanamoto, C. Zhuo, and M. Hashimoto, "A multicore chip load model for pdn analysis considering voltage–current-timing interdependency and operation mode transitions", *IEEE Transactions on Components, Packaging and Manufacturing Technology*, vol. 9, *9*, pp. 1669–1679, 2019.

[29] M. Saint-Laurent and M. Swaminathan, "Impact of power-supply noise on timing in high-frequency microprocessors", *IEEE Transactions on Advanced Packaging*, vol. 27, *1*, pp. 135–144, 2004.

[30] Y. Ogasahara, T. Enami, M. Hashimoto, T. Sato, and T. Onoye, "Validation of a full-chip simulation model for supply noise and delay dependence on average voltage drop with on-chip delay measurement", *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, *10*, pp. 868–872, 2007.

[31] Y. Shim and D. Oh, "System level modeling of timing margin loss due to dynamic supply noise for high-speed clock forwarding interface", *IEEE Transactions on Electromagnetic Compatibility*, vol. 58, *4*, pp. 1349–1358, 2016.

[32] M. Ueno, M. Hashimoto, and T. Onoye, "Real-time on-chip supply voltage sensor and its application to trace-based timing error localization", *in 2015 IEEE 21st International On-Line Testing Symposium (IOLTS)*, 2015, pp. 188–193.

[33] F. Khelil, M. Hamdi, S. Guilley, J. Danger, and N. Selmane, "Fault analysis attack on an FPGA AES implementation", *in NTMS 2008, 2nd International Conference on New Technologies, Mobility and Security, November 5-7, 2008, Tangier, Morocco*, IEEE, 2008, pp. 1–5.

[34] L. Zussa, J.-M. Dutertre, J. Clédière, B. Robisson, and A. Tria, "Investigation of timing constraints violation as a fault injection means", *in 27th Conference on Design of Circuits and Integrated Systems (DCIS)*, Avignon, France, Nov. 2012, pas encore paru.

[35] L. Zussa, J.-M. Dutertre, J. Clédière, and A. Tria, "Power supply glitch induced faults on fpga: an in-depth analysis of the injection mechanism", *in 2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*, 2013, pp. 110–115.

[36] N. Selmane, S. Bhasin, S. Guilley, and J. Danger, "Security evaluation of application-specific integrated circuits and field programmable gate arrays against setup time violation attacks", *IET Inf. Secur.*, vol. 5, *4*, pp. 181–190, 2011.

[37] L. Zussa, J. Dutertre, J. Clédière, and B. Robisson, "Analysis of the fault injection mechanism related to negative and positive power supply glitches using an on-chip voltmeter", *in 2014 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2014, Arlington, VA, USA, May 6-7, 2014*, IEEE Computer Society, 2014, pp. 130–135.

[38] B. Razavi, *Fundamentals of microelectronics*. John Wiley & Sons, 2021.

[39] Z. Kazemi, "Fault Injection Attacks on Embedded Applications : Characterization and Evaluation", Theses, Université Grenoble Alpes [2020-....], Feb. 2022. [Online]. Available: `https://theses.hal.science/tel-03659627`.

[40] O. Bittner, T. Krachenfels, A. Galauner, and J.-P. Seifert, "The forgotten threat of voltage glitching: a case study on nvidia tegra x2 socs", *in 2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, 2021, pp. 86–97.

[41] ARM, *ARM Cortex-A9*, [Accessed: March 24, 2023]. [Online]. Available: `https://developer.arm.com/Processors/Cortex-A9`.

[42] A. Takahashi and M. Tibouchi, "Degenerate fault attacks on elliptic curve parameters in openssl", *in IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*, IEEE, 2019, pp. 371–386.

[43] P. Gallagher, *Digital signature standard (DSS)*. NIST, 2013, FIPS PUB 186–4.

[44] A. Dehbaoui, J. Dutertre, B. Robisson, and A. Tria, "Electromagnetic transient faults injection on a hardware and a software implementations of AES", *in 2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012*, G. Bertoni and B. Gierlichs, Eds., IEEE Computer Society, 2012, pp. 7–15.

[45] P. Bayon, L. Bossuet, A. Aubert, *et al.*, "Contactless electromagnetic active attack on ring oscillator based true random number generator", *in Constructive Side-Channel Analysis and Secure Design*, W. Schindler and S. A. Huss, Eds., 2012, pp. 151–166.

[46] M. Ghodrati, B. Yuce, S. Gujar, C. Deshpande, L. Nazhandali, and P. Schaumont, "Inducing local timing fault through EM injection", *in Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, ACM, 2018, 142:1–142:6.

[47] S. Ordas, L. Guillaume-Sage, K. Tobich, J. Dutertre, and P. Maurine, "Evidence of a larger em-induced fault model", *in Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, M. Joye and A. Moradi, Eds., ser. Lecture Notes in Computer Science, vol. 8968, Springer, 2014, pp. 245–259.

[48] M. Dumont, M. Lisart, and P. Maurine, "Modeling and simulating electromagnetic fault injection", *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 40, *4*, pp. 680–693, 2021.

[49] C. O'Flynn, "Short paper: emfi for safety-critical testing of automotive systems", *in 2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, Los Alamitos, CA, USA: IEEE Computer Society, Sep. 2021, pp. 61–66.

[50] R. Omarouayache, J. Raoult, S. Jarrix, L. Chusseau, and P. Maurine, "Magnetic microprobe design for EM fault attack", *in 2013 International Symposium on Electromagnetic Compatibility*, 2013, pp. 949–954.

[51] A. Dehbaoui, A. Mirbaha, N. Moro, J. Dutertre, and A. Tria, "Electromagnetic glitch on the AES round counter", *in Constructive Side-Channel Analysis and Secure Design - 4th International Workshop, COSADE 2013, Paris, France, March 6-8, 2013, Revised Selected Papers*, E. Prouff, Ed., ser. Lecture Notes in Computer Science, vol. 7864, Springer, 2013, pp. 17–31.

[52] H. Liao and C. Gebotys, "Methodology for EM fault injection: charge-based fault model", *in 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 256–259.

[53] Microchip, *PIC16F687*, [Accessed: March 24, 2023]. [Online]. Available: `https://www.microchip.com/en-us/product/PIC16F687`.

[54] A. Cui and R. Housley, "BADFET: defeating modern secure boot using second-order pulsed electromagnetic fault injection", *in 11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*, W. Enck and C. Mulliner, Eds., USENIX Association, 2017.

[55] H. Martín, T. Korak, E. S. Millán, and M. Hutter, "Fault attacks on strngs: impact of glitches, temperature, and underpowering on randomness", *IEEE Transactions on Information Forensics and Security*, vol. 10, *2*, pp. 266–277, 2015.

[56] D. Ha, K. Woo, S. Meninger, T. Xanthopoulos, E. Crain, and D. Ham, "Time-domain cmos temperature sensors with dual delay-locked loops for microprocessor thermal monitoring", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, *9*, pp. 1590–1601, 2012.

[57] T. Korak, M. Hutter, B. Ege, and L. Batina, "Clock glitch attacks in the presence of heating", *in 2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2014, Busan, South Korea, September 23, 2014*, A. Tria and D. Choi, Eds., IEEE Computer Society, 2014, pp. 104–114.

[58] Atmel Corporation., *Atmel AVR ATmega162 datasheet*, [Accessed: March 24, 2023]. [Online]. Available: `https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2513-8-bit-AVR-Microntroller-ATmega162_Datasheet.pdf`.

[59] M. Hutter and J. Schmidt, "The temperature side channel and heating fault attacks", *IACR Cryptol. ePrint Arch.*, p. 190, 2014.

[60] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine", *in 2003 IEEE Symposium on Security and Privacy (S&P 2003), 11-14 May 2003, Berkeley, CA, USA*, IEEE Computer Society, 2003, pp. 154–165.

[61] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: theory, practice, and countermeasures", *Proc. IEEE*, vol. 100, *11*, pp. 3056–3076, 2012.

[62] A. Baksi, S. Bhasin, J. Breier, D. Jap, and D. Saha, "A survey on fault attacks on symmetric key cryptosystems", *ACM Comput. Surv.*, vol. 55, *4*, 86:1–86:34, 2023.

[63] A. Beckers, S. Guilley, P. Maurine, C. O'Flynn, and S. Picek, "(adversarial) electromagnetic disturbance in the industry", *IEEE Trans. Computers*, vol. 72, *2*, pp. 414–422, 2023.

[64] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "CLKSCREW: exposing the perils of security-oblivious energy management", *in 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, E. Kirda and T. Ristenpart, Eds., USENIX Association, 2017, pp. 1057–1074.

[65] ARM, *ARM Trustzone*, [Accessed: March 29, 2023]. [Online]. Available: `https://www.arm.com/technologies/trustzone-for-cortex-a`.

[66] P. Qiu, D. Wang, Y. Lyu, R. Tian, C. Wang, and G. Qu, "Voltjockey: A new dynamic voltage scaling-based fault injection attack on intel SGX", *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 40, *6*, pp. 1130–1143, 2021.

[67] Intel, *Intel SGX Enclave*, [Accessed: March 24, 2023]. [Online]. Available: `https://www.intel.com/content/dam/develop/external/us/en/documents/overview-of-intel-sgx-enclave-637284.pdf`.

[68] Z. Chen, G. Vasilakis, K. Murdock, E. Dean, D. F. Oswald, and F. D. Garcia, "Voltpillager: hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface", *in 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds., USENIX Association, 2021, pp. 699–716.

[69] R. Buhren, H. N. Jacob, T. Krachenfels, and J. Seifert, "One glitch to rule them all: fault injection attacks against amd's secure encrypted virtualization", *in CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds., ACM, 2021, pp. 2875–2889.

[70] AMD, *AMD Secure Encrypted Virtualization (SEV)*, [Accessed: March 24, 2023]. [Online]. Available: `https://www.amd.com/en/developer/sev.html#:~:text=AMD%5C%20Secure%5C%20Encrypted%5C%20Virtualization%5C%2DEncrypted,to%5C%20a%5C%20CPU%5C%20register%5C%20state.`.

[71] PJRC, *Teensy 4.0 development board.* [Accessed: March 31, 2023]. [Online]. Available: `https://www.pjrc.com/store/teensy40.html`.

[72] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller", *in 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2013, pp. 77–88.

[73] J. Yiu, *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors*. Newnes, 2013.

[74] L. Rivière, Z. Najm, P. Rauzy, J. Danger, J. Bringer, and L. Sauvage, "High precision fault injections on the instruction cache of armv7-m architectures", *in IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015*, IEEE Computer Society, 2015, pp. 62–67.

[75] O. Trabelsi Ltci, L. Sauvage Ltci, and J.-L. Danger Ltci, "Characterization of electromagnetic fault injection on a 32-bit microcontroller instruction buffer", *in 2020 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2020, pp. 1–6.

[76] L. Dureuil, M. Potet, P. de Choudens, C. Dumas, and J. Clédière, "From code review to fault injection attacks: filling the gap using fault model inference", *in Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, N. Homma and M. Medwed, Eds., ser. Lecture Notes in Computer Science, vol. 9514, Springer, 2015, pp. 107–124.

[77] ARM Limited, *Armv7-m architecture reference manual.* [Accessed: February 22, 2022]. [Online]. Available: `https://developer.arm.com/documentation/ddi0403/latest`.

[78] V. Werner, L. Maingault, and M. Potet, "An end-to-end approach for multi-fault attack vulnerability assessment", *in Workshop on Fault Detection and Tolerance in Cryptography*, Milan, Italy: IEEE, 2020, pp. 10–17.

[79] N. Timmers, A. Spruyt, and M. Witteman, "Controlling PC on ARM Using Fault Injection", *in 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2016, pp. 25–35.

[80] ARM, *ARMv7-A Architecture Reference Manual.* [Accessed: April 10, 2023]. [Online]. Available: `https://developer.arm.com/documentation/ddi0406/latest/`.

[81] M. S. Kelly, K. Mayes, and J. F. Walker, "Characterising a CPU fault attack model via run-time data analysis", *in 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2017, pp. 79–84.

[82] Atmel Corporation., *Attiny841 datasheet.* [Accessed: April 6, 2023]. [Online]. Available: `https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8495-8-bit-AVR-Microcontrollers-ATtiny441-ATtiny841_Datasheet.pdf`.

[83] T. Given-Wilson, N. Jafri, J. Lanet, and A. Legay, "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 Computer Society, 2017, pp. 293–300.

[84] A. Bogdanov, L. R. Knudsen, G. Leander, *et al.*, "Present: an ultra-lightweight block cipher", *in Cryptographic Hardware and Embedded Systems - CHES 2007*, P. Paillier and I. Verbauwhede, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 450–466.

[85] L. R. Knudsen and G. Leander, "Present – block cipher", *in Encyclopedia of Cryptography and Security*, H. C. A. van Tilborg and S. Jajodia, Eds. Boston, MA: Springer US, 2011, pp. 953–955.

[86] A. Menu, J. Dutertre, O. Potin, J. Rigaud, and J. Danger, "Experimental analysis of the electromagnetic instruction skip fault model", *in 15th Design & Technology of Integrated Systems in Nanoscale Era, DTIS 2020, Marrakech, Morocco, April 1-3, 2020*, IEEE, 2020, pp. 1–7.

[87] MICROCHIP., *ATmega328P*, [Accessed: April 7, 2023]. [Online]. Available: `https://www.microchip.com/en-us/product/ATmega328P`.

[88] T. Trouchkine, G. Bouffard, and J. Clédière, "EM fault model characterization on socs: from different architectures to the same fault model", *in 2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, IEEE, 2021, pp. 31–38.

[89] ARM, *ARM Cortex-A53*, [Accessed: April 7, 2023]. [Online]. Available: `https://developer.arm.com/Processors/Cortex-A53`.

[90] Intel Corporation, *Intel Core i3-6100T Processor*, [Accessed: April 7, 2023]. [Online]. Available: `https://ark.intel.com/content/www/us/en/ark/products/90734/intel-core-i36100t-processor-3m-cache-3-20-ghz.html`.

[91] OpenSSL, *OpenSSL Cryptography and SSL/TLS Toolkit*, [Accessed: April 7, 2023]. [Online]. Available: `https://www.openssl.org/source/gitrepo.html`.

[92] V. Khuat, J.-L. Danger, and J.-M. Dutertre, "Laser fault injection in a 32-bit microcontroller: from the flash interface to the execution pipeline", *in 2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, 2021, pp. 74–85.

[93] ARM, *ARM Cortex-M0+*, [Accessed: April 7, 2023]. [Online]. Available: `https://developer.arm.com/Processors/Cortex-M0-Plus`.

[94] J. Laurent, V. Beroulle, C. Deleuze, F. Pebay-Peyroula, and A. Papadimitriou, "Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a RISC-V processor", *Microprocessors and Microsystems*, vol. 71, 2019.

[95] J. Laurent, C. Deleuze, F. Pebay-Peyroula, and V. Beroulle, "Bridging the gap between RTL and software fault injection", *ACM J. Emerg. Technol. Comput. Syst.*, vol. 17, *3*, 38:1–38:24, 2021.

[96] RISC-V Foundation, *The RISC-V instruction set manual*, [Accessed: May 4, 2021]. [Online]. Available: `https://riscv.org/technical/specifications/`.

[97] S. Tollec, M. Asavoae, D. Couroussé, K. Heydemann, and M. Jan, "Exploration of fault effects on formal RISC-V microarchitecture models", *in Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2022, Virtual Event / Italy, September 16, 2022*, IEEE, 2022, pp. 73–83.

[98] J.-M. Dutertre, V. Beroulle, P. Candelier, *et al.*, "Laser fault injection at the cmos 28 nm technology node: an analysis of the fault model", *in 2018 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2018, pp. 1–6.

[99] C. O'Flynn and Z. (Chen, "Chipwhisperer: an open-source platform for hardware embedded security research", *in International Workshop on Constructive Side-Channel Analysis and Secure Design*, E. Prouff, Ed., ser. Lecture Notes in Computer Science, vol. 8622, Paris, France: Springer, 2014, pp. 243–260.

[100] J. Yiu, *The Definitive Guide to ARM Cortex-M0 and Cortex-M0+ Processors*. Newnes, 2015.

[101] ARM Limited, *ARMv6-M Architecture Reference Manual*, [Accessed: November 22, 2021]. [Online]. Available: `https://developer.arm.com/documentation/ddi0419/c?lang=en`.

[102] ARM Limited, *ARM Architecture Reference Manual Thumb-2 Supplement*, [Accessed: May 4, 2021]. [Online]. Available: `https://developer.arm.com/documentation/ddi0308/d`.

[103] I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle, and P. Maistri, "Microarchitecture-aware fault models: experimental evidence and cross-layer inference methodology", *in 2021 16th International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2021, pp. 1–6.

[104] N. Theißing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, "Comprehensive analysis of software countermeasures against fault attacks", *in 2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 404–409.

[105] J. Laurent, "Modélisation de fautes utilisant la description RTL de microarchitectures pour l'analyse de vulnérabilité conjointe matérielle-logicielle", Theses, Université Grenoble Alpes [2020-....], Nov. 2020.

[106] ARM Limited, *ARM Cortex-M3 DesignStart Eval RTL and FPGA Quick Start Guider0p0*, [Accessed: November 29, 2021]. [Online]. Available: https://developer.arm.com/documentation/100895/0000/introduction/what-is-cortex-m3-designstart-eval-?lang=en.

[107] ARM Limited, *ARM Cortex-M3 DesignStart Eval RTL and Testbench User Guide r0p0*, [Accessed: November 29, 2021]. [Online]. Available: https://developer.arm.com/documentation/100894/0000/technical-overview/example-system.

[108] H. Pan, "High performance, variable-length instruction encodings", Ph.D. dissertation, Massachusetts Institute of Technology, 2002.

[109] Prasad Kulkarni, *16/32-Bit ARM-Thumb Architecture and AX Extensions.* [Accessed: March 2, 2022]. [Online]. Available: http://www.ittc.ku.edu/~kulkarni/research/thumb%5C_ax.pdf.

[110] MIPS Technologies, Inc., *microMIPSTM Instruction Set Architecture Uncompromised Performance, Minimum System Cost.* [Accessed: March 2, 2022]. [Online]. Available: https://s3-eu-west-1.amazonaws.com/downloads-mips/mips-documentation/login-required/micromips%5C_instruction%20%5C_set%5C_architecture.pdf.

[111] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V compressed instruction set manual, version 1.7", *EECS Department, University of California, Berkeley, UCB/EECS-2015-157*, 2015.

[112] informIT, *Understanding ARM architectures.* [Accessed: March 1, 2022]. [Online]. Available: https://www.informit.com/articles/article.aspx?p=1620207%5C&seq%20Num=3.

[113] Tom Shanley | Mindshare, Inc., *X86 instruction set architecture.* [Accessed: March 2, 2022]. [Online]. Available: https://www.mindshare.com/files/ebooks/x86%5C%20Instruction%5C%20Set%5C%20Architecture.pdf.

[114] MIPS Technologies, Inc., *MIPS32™ Architecture For Programmers Volume II: The MIPS32™ Instruction Set*. [Accessed: March 2, 2022]. [Online]. Available: https://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS%20%5C_Vol2.pdf.

[115] MIPS Technologies, Inc., *Mips64™ architecture for programmers volume ii: the mips64™ instruction set.* [Accessed: March 2, 2022]. [Online]. Available: https://scc.ustc.edu.cn/zlsc/lxwycj/200910/%20W020100308600769158777%20.pdf.

[116] 2CS Division, EECS Department, University of California, Berkeley, "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA", 2019. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[117] Intel Corporation, *Intel®64 and IA-32 Architectures Software Developer Manuals, Volume 3A: System Programming Guide, Part 1*. Santa Clara, CA: Intel Corporation, 2016. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf.

[118] Advanced Micro Devices, Inc., *AMD64 Architecture Programmer's Manual Volumes 1–5*. Santa Clara, CA: Advanced Micro Devices, Inc., 2023. [Online]. Available: https://www.amd.com/system/files/TechDocs/40332.pdf.

[119] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto, "Static analysis of seu effects on software applications", *in Proceedings. International Test Conference*, 2002, pp. 500–508.

[120] M. Escouteloup, R. Lashermes, J.-L. Lanet, and J. J.-A. Fournier, "Recommendations for a radically secure ISA", *in CARRV 2020 - Workshop on Computer Architecture Research with RISC-V*, Valence (virtual), Spain: ACM, May 2020, pp. 1–22.

[121] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: systems, languages, and applications", *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, *1*, pp. 1–34, 2012.

[122] J. Gratchoff, N. Timmers, A. Spruyt, and L. Chmielewski, "Proving the wild jungle jump", Technical report, University of Amsterdam, Tech. Rep., 2015.

[123] M. Agoyan, J.-M. Dutertre, D. Naccache, B. Robisson, and A. Tria, "When clocks fail: on critical paths and clock faults", *in Smart Card Research and Advanced Application*, D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 182–193.

[124] Y. Li, K. Ohta, and K. Sakiyama, "New fault-based side-channel attack using fault sensitivity", *IEEE Transactions on Information Forensics and Security*, vol. 7, *1*, pp. 88–97, 2012.

[125] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model", *in International Workshop on Cryptographic Hardware and Embedded Systems*, M. Joye and J. Quisquater, Eds., ser. Lecture Notes in Computer Science, vol. 3156, Cambridge, MA, USA: Springer, Aug. 2004, pp. 16–29.

[126] M. Alioto, M. Poli, and S. Rocchi, "Differential power analysis attacks to precharged buses: A general analysis for symmetric-key cryptographic algorithms", *IEEE Transactions on Dependable and Secure Computing*, vol. 7, *3*, pp. 226–239, 2010.

[127] M. Randolph and W. Diehl, "Power side-channel attack analysis: A review of 20 years of study for the layman", *Cryptography*, vol. 4, *2*, p. 15, 2020.

[128] M. A. Shelton, N. Samwel, L. Batina, F. Regazzoni, M. Wagner, and Y. Yarom, "Rosita: towards automatic elimination of power-analysis leakage in ciphers", *in Annual Network and Distributed System Security Symposium*, Virtual event: The Internet Society, Feb. 2021.

[129] E. Boespflug, C. Ene, L. Mounier, and M. Potet, "Countermeasures optimization in multiple fault-injection context", *in 17th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2020, Milan, Italy, September 13, 2020*, IEEE, 2020, pp. 26–34.

[130] E. Boespflug, A. Bouguern, L. Mounier, and M. Potet, "A tool assisted methodology to harden programs against multi-faults injections", *CoRR*, vol. abs/2303.01885, 2023.

[131] I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle, and P. Maistri, "Variable-length instruction set: feature or bug?", *in 25th Euromicro Conference on Digital System Design*, Maspalomas, Spain: IEEE, Aug. 2022, pp. 464–471.

[132] I. Alshaer, B. Colombier, C. Deleuze, P. Maistri, and V. Beroulle, "Cross-layer inference methodology for microarchitecture-aware fault models", *Microelectronics Reliability*, vol. 139, p. 114 841, 2022, ISSN: 0026-2714.

LIST OF FIGURES

LIST OF TABLES

SOMMAIRE

L'utilisation des dispositifs des systèmes embarqués connaît une croissance rapide dans divers domaines de la vie. Par exemple, selon les prévisions, le nombre d'appareils Internet des objets (IoT) en utilisation dans le monde devrait atteindre environ 30 milliards d'ici 2030 [1], comme illustré dans la Figure 5.4. Cependant, la complexité de ces dispositifs, ainsi que de leurs applications en cours d'exécution, ne cesse d'augmenter. Cela ouvre la porte à deux considérations : le besoin de hautes performances et de nouvelles méthodes pour faire face à de telles avancées, et d'autre part, l'émergence de nouvelles vulnérabilités exploitables par des attaquants à différents niveaux. Étant donné que des données sensibles sont fréquemment traitées par les systèmes embarqués, une forme de protection est nécessaire pour prévenir toute fuite ou modification d'informations. Le traitement et la protection réels peuvent être vulnérables aux attaques visant à extraire ces informations sensibles. Les attaques physiques en particulier constituent une menace sérieuse pour les systèmes embarqués.

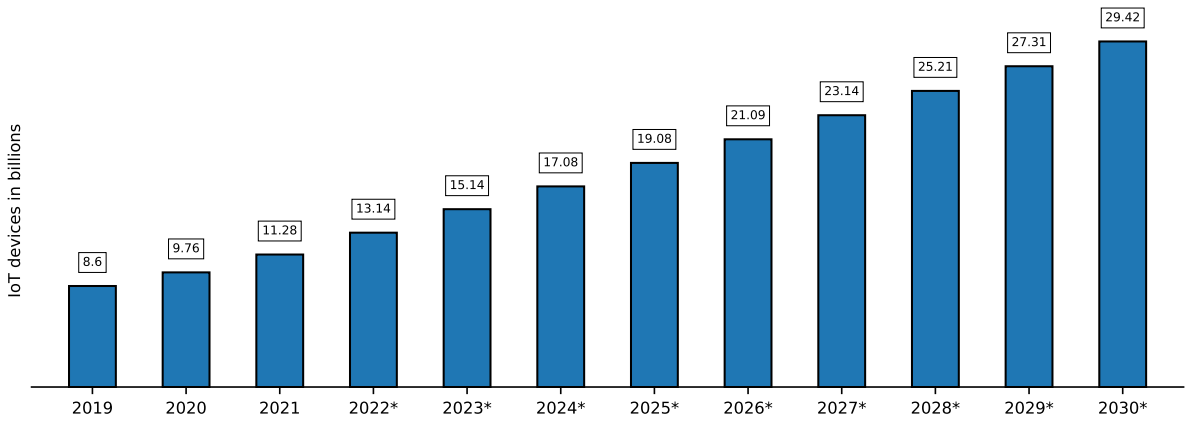


Figure 5.4: Nombre d'appareils IoT dans le monde 2019-2021, avec prévisions jusqu'en 2030 ([1]).

Dans le contexte de la sécurité matérielle, les attaques physiques désignent diverses techniques et méthodes visant à compromettre la sécurité des dispositifs numériques. Ces attaques exploitent les vulnérabilités dans les propriétés physiques ou la mise en

œuvre matérielle du dispositif pour supprimer, modifier, obtenir ou empêcher l'accès à des données confidentielles.

Les attaques physiques les plus courantes sont les attaques par canaux auxiliaires et les attaques par injection de fautes. Les attaques par canaux auxiliaires sont des attaques physiques passives visant principalement à exploiter la fuite involontaire d'informations à partir des caractéristiques physiques d'un dispositif, telles que la consommation d'énergie [2], les émissions électromagnétiques [3] ou les informations de synchronisation [4]. En capturant et en analysant ces signaux de canaux auxiliaires, les attaquants peuvent déduire des informations sensibles, telles que des clés de chiffrement.

Les attaques par injection de fautes, en revanche, sont des attaques physiques actives, potentiellement non invasives, où l'attaquant tentera intentionnellement de modifier le comportement normal d'un dispositif pendant l'exécution du programme en provoquant une ou plusieurs fautes, puis en observant le comportement erroné. Les fautes résultantes pourraient révéler un comportement intéressant pouvant être exploité davantage en tant que vulnérabilité. Les attaques par injection de fautes sont devenues un sujet de recherche attrayant depuis l'attaque bien connue de Boneh *et al.* [5], où ils ont réussi à compromettre certains protocoles cryptographiques en induisant des fautes dans les calculs.

Pour injecter une faute, une interférence physique est appliquée sur le dispositif numérique : radiations [6], lumière laser [7], impulsions électromagnétiques [8], variations de l'alimentation électrique [9], perturbations du signal d'horloge [10], ou changements des conditions environnementales tels que la température [11] ou autre. De plus, des études récentes [12]–[14] ont démontré la capacité à effectuer des attaques par injection de fautes à distance en utilisant des logiciels pour manipuler les régulateurs de tension et/ou les systèmes de gestion de l'énergie dans les dispositifs modernes. Cela a suscité davantage d'attention pour les attaques par injection de fautes.

Afin d'analyser les vulnérabilités pouvant être exploitées à l'aide d'attaques par injection de fautes et de proposer des contre-mesures efficaces, les évaluateurs, les développeurs et les concepteurs ont besoin de modèles de fautes précis. Ces modèles de fautes servent de représentations abstraites des effets réels provoqués par les fautes et sont construits grâce à l'analyse et à la caractérisation de ces effets à différents niveaux des systèmes numériques. Il est crucial de garantir une caractérisation

et une compréhension appropriées des effets de l'injection de fautes pour éviter des modèles de fautes incomplets. Le non-respect de cette exigence peut entraîner soit une sous-ingénierie, laissant des menaces persistantes en matière de sécurité, soit une sur-ingénierie, entraînant des coûts inutiles et potentiellement une dégradation des performances.

L'objectif principal de cette thèse est d'effectuer une analyse inter-couches pour examiner les effets de l'injection de fautes, permettant une meilleure compréhension de ces effets aux niveaux logiciel et matériel. En conséquence, la recherche proposera des modèles de fautes réalistes, explicables et dignes de confiance à différents niveaux d'abstraction du système. Ces modèles de fautes faciliteront les processus d'analyse de vulnérabilités complets et permettront la conception et le développement efficaces de contre-mesures.

Contributions

Tout d'abord, cette thèse présente une étude des techniques d'injection de fautes qui utilisent les violations de temps pour induire des fautes dans les systèmes numériques. Elle comprend également des exemples d'attaques réelles correspondant à chaque technique. En outre, la thèse examine l'état actuel des connaissances en termes de caractérisation et de modélisation des effets des attaques par injection de fautes. Cette analyse révèle que les études précédentes se sont principalement concentrées sur les aspects logiciels ou matériels séparément. Bien que certaines études aient tenté de combler le fossé entre les deux niveaux, elles n'ont effectué des simulations qu'au niveau de l'architecture du jeu d'instructions (ISA) et du transfert de registre (RTL), sans valider leur analyse par des injections de fautes physiques.

De plus, la thèse offre des preuves expérimentales qui démontrent les limites de la caractérisation et de la modélisation des effets de l'injection de fautes basées sur un seul niveau d'analyse. En conséquence, la thèse propose une approche globale pour l'analyse des fautes inter-couches, visant à établir des modèles de fautes fiables à différents niveaux d'abstraction. Elle introduit également des mesures pour évaluer l'efficacité des modèles de fautes proposés.

En outre, la mise en œuvre de la méthodologie proposée, tout en impliquant des injections de fautes et des simulations sur divers processeurs Arm Cortex-M, a permis de déduire des modèles de fautes réalistes au niveau de l'encodage binaire des instruc-

tions : “Skip”, “Skip and repeat”, et “Skip and repeat non séquentiel” pour un nombre spécifique de bits. Ces modèles de fautes permettent d'expliquer un large éventail de comportements défectueux obtenus à des niveaux d'abstraction plus élevés, y compris aux niveaux de l'assemblage et de l'application, indépendamment des instructions cibles et du dispositif cible. Les explications fournies sont également applicables à un large éventail de comportements défectueux observés qui ont été documentés dans la littérature. Sur la base des modèles proposés, la thèse fournit des exemples d'exploitation et d'analyse de vulnérabilité. Ceci démontre la haute fidélité des modèles de fautes proposés. En outre, un outil permettant de simuler ces modèles est également présenté.

De même, l'analyse des effets des fautes à des niveaux d'abstraction inférieurs a permis de dériver des modèles de fautes fiables au niveau RTL : *anticipate the update* et *prevent the update* de la valeur d'un registre à un cycle d'horloge donné. En utilisant ces modèles de fautes, il devient possible d'observer des fautes qui sont identiques aux comportements défectueux obtenus par injection de fautes physiques, ou en simulant les modèles de fautes de codage binaire susmentionnés. En outre, la compréhension des effets de l'injection de fautes au niveau RTL a conduit à la déduction d'un nouveau modèle de fautes connu sous le nom de modèle de fautes *partial update*. Ce modèle offre une augmentation significative de la capacité à expliquer les comportements défectueux.

Enfin, la thèse ouvre des pistes pour diverses directions de recherche. Il s'agit notamment d'utiliser les modèles de fautes proposés pour évaluer les vulnérabilités des logiciels sur la base de propriétés de sécurité prédéfinies. En outre, la thèse démontre la faisabilité du développement de contre-mesures rentables en comprenant les effets potentiels de l'injection de fautes. En outre, la thèse met en évidence la capacité de combiner des comportements défectueux par l'injection de multiples glitches. Elle met également en évidence l'applicabilité des modèles de fautes lors de l'utilisation de différentes techniques d'injection de fautes.

Dans ce qui suit, un résumé de chaque chapitre de la thèse est présenté.

État de l'art

Ce chapitre a présenté les contraintes de synchronisation dans une conception numérique simple. Ensuite, il a démontré différentes techniques d'injection de fautes pouvant en-

traîner la violation de ces contraintes, résultant en divers comportements défectueux. Plusieurs exemples donnés ont illustré comment ces comportements défectueux résultants pourraient être utilisés pour mener des attaques nuisibles.

Après cela, ce chapitre a passé en revue l'état de l'art en termes de caractérisation, d'analyse et de modélisation des effets de l'injection de fautes. Il a été montré que de nombreuses études décrivaient l'effet de l'injection de fautes comme des fautes aléatoires de bits ou d'octets. En revanche, plusieurs études se sont concentrées sur la caractérisation de l'effet de l'injection de fautes uniquement à un seul niveau d'abstraction, spécifiquement au niveau de l'ISA (architecture de jeu d'instructions). Néanmoins, certaines de ces études ont tenté de fournir une analyse supplémentaire au niveau micro-architectural en tenant compte du composant architectural défectueux ou du chemin de propagation de la faute. Cependant, toutes ces études ont proposé des modèles de fautes plutôt généraux, tels que le saut d'instruction et la corruption d'instructions. Ces modèles de fautes ne sont clairement pas suffisants pour évaluer les vulnérabilités des codes logiciels ou des conceptions matérielles. De plus, ils conduiraient à développer ou à concevoir des contre-mesures non optimales. Cela affecterait certainement soit le coût, les performances, ou la sécurité du dispositif.

Quelques études, en revanche, ont combiné des niveaux d'abstraction élevés et bas dans le but de promouvoir la compréhension de l'effet de la propagation de la faute dans un système numérique. Cependant, elles n'ont réalisé que des simulations de fautes sur un processeur RISC-V et aucune injection physique n'a été effectuée. En incorporant des injections de fautes physiques, le réalisme et la fiabilité des modèles proposés peuvent être mieux évalués, et les résultats peuvent être plus applicables à un éventail plus large d'architectures de systèmes.

Enfin, d'autres travaux ont tenté de se concentrer sur leur analyse au plus bas niveau d'abstraction en décrivant comment la faute induite affecte le comportement normal des transistors.

Le besoin d'une analyse multi-niveaux et d'une méthodologie proposée

Dans ce chapitre, nous avons présenté les problèmes existants dans l'analyse et la compréhension des attaques de fautes dans les microarchitectures complexes, en

nous concentrant uniquement sur le niveau ISA pour l'analyse. Nous avons mis cela en évidence en fournissant des preuves expérimentales de fautes intrinsèquement microarchitecturales, en utilisant le glitch d'horloge comme technique d'injection de fautes. Les résultats expérimentaux ont montré que les comportements défectueux observés peuvent dépendre de la cible, du prologue et de l'architecture.

Ensuite, nous avons proposé une nouvelle méthodologie pour fournir une analyse inter-couches afin de caractériser les comportements défectueux. Cette méthodologie peut être utilisée pour construire des modèles de fautes réalistes à différents niveaux, tels que les niveaux RTL et logiciel. Elle peut également fournir une explication sur l'origine des défauts observés. Cela permet donc de concevoir des contre-mesures adaptées au coût le plus approprié au niveau du matériel et du logiciel. En outre, cela facilite le processus d'analyse des vulnérabilités.

Enfin, des mesures permettant d'évaluer le réalisme et la qualité des modèles de défaillance proposés sont fournies. Ces mesures sont les suivantes: couverture, fidélité, et complexité.

Simulation RTL préliminaire et nouveaux modèles de défauts de codage binaire

Ce chapitre est introduit par une série d'expériences de simulation de fautes matérielles qui ont fourni des connaissances sur l'origine de certains des comportements défectueux observés au niveau micro-architectural. Plus important encore, ces expériences ont révélé la logique derrière différents comportements défectueux observés lors de l'application d'une petite modification au programme cible ; cela est dû à l'ensemble d'instructions Thumb2 qui prend en charge des instructions de longueur variable, ce qui peut entraîner un code aligné ou mal aligné en mémoire. Ainsi, le processeur peut récupérer des instructions alignées ou mal alignées. C'était la connaissance fondamentale pour déduire de nouveaux modèles de fautes au niveau de l'encodage binaire des instructions. Les principaux modèles déduits sont "Skip" et "Skip and repeat" d'un nombre spécifique de bits. Ce nombre de bits est lié à la taille d'accès à la mémoire flash, à la taille de la ligne de cache, au cache d'instructions ou à la taille du registre dans le chemin de récupération. De toute évidence, ces modèles offraient des explications pour la plupart des comportements défectueux observés. De

plus, leur réalisme a été prouvé en ciblant différentes instructions et en utilisant divers dispositifs cibles. De plus, des exemples d'exploitation et d'analyse de vulnérabilité en utilisant les modèles proposés ont été réalisés et validés expérimentalement. Enfin, ce chapitre a décrit un outil d'automatisation pour simuler ces modèles et effectuer la comparaison entre les différentes résultats d'injection et de simulation.

Simulation de défauts matériels et modèle de défauts de mise à jour partielle

Ce chapitre a poursuivi l'approche d'analyse inter-couches en couvrant davantage de niveaux d'abstraction. En particulier, il est descendu au niveau du système et a pris en compte le niveau matériel pour effectuer la simulation de fautes RTL (Register-Transfer Level). De nouveaux modèles de fautes RTL ont été proposés. Ces modèles ont permis d'observer des fautes identiques aux comportements défectueux observés lors de l'injection physique de fautes. De plus, une approche de simulation de fautes basée sur l'analyse du chemin critique a été présentée. Cette approche a accéléré le processus de simulation de fautes et a révélé l'origine des comportements défectueux observés.

Grâce à l'analyse RTL, un nouveau modèle de faute au niveau logiciel a été déduit : le modèle de faute "Partial update", qui est également applicable au niveau RTL. Ce modèle se décline en deux sous-cas : "Partial update from the precharge value" et "Partial update from the previous value". Ces modèles de fautes permettent d'expliquer un large éventail de comportements défectueux obtenus lors de campagnes d'injection de fautes par horloge glitch et qui étaient auparavant inexpliqués. Par conséquent, ils peuvent être utilisés pour effectuer une analyse de vulnérabilité des codes logiciels contre les attaques par faute et contribuer à une meilleure conception de contre-mesures efficaces et peu coûteuses. Cependant, ils sont plus complexes que les modèles de fautes précédemment déduits : "Skip" et "Skip and repeat".

Autres résultats et détails

Dans ce chapitre, des analyses supplémentaires, des résultats et des détails sont fournis. De plus, de nouvelles orientations de recherche peuvent être suivies en fonction du

travail présenté dans ce chapitre. Tout d'abord, différents scénarios pour compromettre l'intégrité du flux de contrôle d'un programme sont présentés. Ces scénarios montrent comment un évaluateur de sécurité peut prédire un comportement défectueux en se basant sur les modèles de fautes proposés. De plus, les résultats expérimentaux ont prouvé le réalisme de cette prédiction et, par conséquent, la fiabilité des modèles de fautes proposés. De plus, la compréhension de l'effet possible de la faute a permis de protéger le code avec des contre-mesures simples sans surcharge supplémentaire.

En outre, l'idée du cheval de Troie a été présentée, ce qui a également renforcé la fiabilité des modèles de fautes proposés. De plus, cela montre un nouveau modèle de menace résultant de l'injection de fautes, dans lequel le développeur de logiciels ou le compilateur utilisé représente l'adversaire.

De plus, ce chapitre montre l'applicabilité de l'injection de plusieurs glitches pour combiner des comportements défectueux, chacun d'eux pouvant être obtenu en résultat d'un seul glitch. Cela augmente les capacités d'un attaquant à obtenir des fautes exploitables. En même temps, cela complexifie la perspective de la protection. La recherche de nouveaux effets défectueux résultant de l'injection de plusieurs fautes serait une perspective intéressante de ce travail.

Enfin, la réalisation de campagnes d'injection de fautes par glitch de tension a validé l'applicabilité des modèles de fautes proposés lors de la réalisation de différentes techniques d'injection de fautes par rapport au glitch d'horloge. Il a été démontré que dans certaines campagnes, la couverture atteignait 100 %. Cela confirme la haute fidélité et la solidité des modèles de fautes proposés, même lorsque l'on utilise une technique d'injection distincte. La possibilité d'observer des comportements défectueux à l'aide de glitches de tension distincts des comportements défectueux observés avec des glitches d'horloge pourrait être explorée dans une étude future.

Conclusion

À mesure que les systèmes numériques deviennent de plus en plus courants et que le marché de l'Internet des objets (IoT) connaît une croissance significative, il y a eu une attention accrue portée à la sécurité parmi les concepteurs matériels et les développeurs de logiciels. L'objectif est de protéger ces dispositifs contre les menaces potentielles. Parmi les diverses préoccupations en matière de sécurité, les attaques physiques représentent un risque significatif, l'injection de fautes étant une forme ma-

jeure de telles attaques.

Afin d'évaluer les systèmes numériques contre les attaques par injection de fautes et de les protéger contre de telles menaces, les développeurs de logiciels et les concepteurs matériels doivent s'appuyer sur des modèles de fautes réalistes. Cependant, la complexité des microprocesseurs embarqués et leur comportement sous ces attaques posent d'énormes défis lorsqu'il s'agit de dériver des modèles de fautes précis basés sur des observations limitées de microprocesseurs défectueux, en particulier lorsque l'analyse est limitée à un seul niveau d'abstraction. De plus, le recours à des modèles de fautes peu pratiques ou aléatoires conduirait à une analyse de vulnérabilité inadéquate et inexacte, pouvant potentiellement entraîner le développement ou la conception de contre-mesures soit sur-ou sous-dimensionnées.

L'objectif de cette thèse était de proposer une approche d'analyse inter-couches pour mieux analyser, caractériser et comprendre les effets des attaques par injection de fautes à différents niveaux d'abstraction d'un système numérique. Par conséquent, des modèles de fautes logiciels et matériels dignes de confiance et fiables ont été présentés, permettant des processus complets d'analyse de vulnérabilité ainsi qu'une conception et/ou un développement efficaces de contre-mesures.

Vous trouverez les publications et les communications basées sur ces travaux en cliquant sur : Publications and communications