Fault Attacks on Embedded Software: New Directions in Modeling, Design, and Mitigation

Bilgiday Yuce

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Patrick R. Schaumont, Chair
Michael S. Hsiao
Leyla Nazhandali
Cameron D. Patterson
Danfeng Yao

Dec 4, 2017
Blacksburg, Virginia

Keywords: Embedded Systems, Fault Attacks,Countermeasures, Fault Models,
Fault Simulation, Fault Mitigation

Fault Attacks on Embedded Software: New Directions in Modeling, Design, and Mitigation

Bilgiday Yuce

(ABSTRACT)

This research investigates an important class of hardware attacks against embedded software, which uses fault injection as a hacking tool. Fault attacks use well-chosen, targeted fault injection combined with clever system response analysis to break the security of a system.

In case of a fault attack on embedded software, faults are injected into the underlying processor hardware and their effects are observed in the executed software's output. This introduces an additional difficulty in mitigation of fault attack risk. Designing efficient countermeasures requires first understanding software, instruction-set, and hardware level components of fault attacks, and then, systematically addressing the vulnerabilities at each level.

This research first proposes an instruction fault sensitivity model to capture effects of fault injection on embedded software. Based on the instruction fault sensitivity model, a novel fault attack method called MAFIA (Micro-architecture Aware Fault Injection Attack) is also introduced. MAFIA exploits the vulnerabilities in multiple abstraction layers. This enables an adversary to determine best points to attack during the execution as well as pinpoint the desired fault effects. It has been shown that MAFIA breaks the existing countermeasures with significantly fewer fault injections than the traditional fault attacks.

Another contribution of the research is a fault attack simulator, MESS (Micro-architectural Embedded System Simulator). MESS enables a user to model hardware, instruction-set, and software level components of fault attacks in a simulation environment. Thus, software designers can use MESS to evaluate their programs against several real-world fault attack scenarios.

The final contribution of this research is the fault-attack-resistant FAME (Fault-attack Aware Microprocessor Extensions) processor, which is suited for embedded, constrained systems. FAME combines fault detection in hardware and fault response in software. This allows low-cost, performance-efficient, flexible, and backward-compatible integration of hardware and software techniques to mitigate fault attack risk. FAME has been designed as an architectural concept as well as implemented as a chip prototype. In addition to protection mechanisms, the chip prototype also includes fault injection and analysis features to ease fault attack research.

The findings of this research indicate that considering multiple abstraction layers together is essential for efficient fault attacks, countermeasures, and evaluation techniques.

Fault Attacks on Embedded Software: New Directions in Modeling, Design, and Mitigation

Bilgiday Yuce

(GENERAL AUDIENCE ABSTRACT)

Today, we trust a range of embedded computers to process and protect our sensitive data. For instance, credit cards process sensitive financial data during electronic payment. Similarly, smartphones use and store private user data. This research investigates fault attacks, a serious threat to the security of embedded computers.

In a fault attack, an adversary breaches the security by injecting intentional faults in an embedded computer. To induce faults, the adversary deliberately manipulates the operating conditions of the computer such as the supply voltage and ambient temperature. These faults interfere with the correct operation of the computer and cause temporary malfunctions in its hardware. The adversary then exploits the malfunctions to break the security.

Although fault injection is a powerful hacking tool that may affect any security mechanism, there is no generic technique to deal with the security threat of faults. This research seeks a broader, deeper understanding of fault attacks and appropriate countermeasures for them. Our contributions include a novel fault modeling method, efficient fault attacks, a fault attack simulator, and a low-cost fault-attack-aware microprocessor. This research also provides a deeper understanding of causes and effects of faults, which can be utilized in the design of fault attacks, countermeasures, and metrics.

Acknowledgments

First, I would like to sincerely thank my advisor Prof. Patrick Schaumont for his time, help, and support. His integrity and meticulousness have greatly influenced me as a researcher and an individual. This work would not have been possible without his continuous encouragement, guidance, and enthusiasm for my research.

I acknowledge my committee members Prof. Leyla Nazhandali, Prof. Michael Hsiao, Prof. Cameron Patterson, and Prof. Danfeng Yao, for their valuable feedback on my dissertation. In addition, I express special thanks to Dr. Nazhandali for her help, support, and fruitful discussions; she has been like a co-advisor for me throughout my PhD journey.

I also deeply appreciate the help from the members and alumni of Secure Embedded System Group. Thank you Chinmay Deshpande, Dr. Nahid Farhady Ghalaty, Dr. Aydin Aysu, Ege Gulcan, Marjan Ghodrati, Conor Patrick, Abhishek Ajey Bendre, Yuan Yao, Archanaa Santhana Krishnan, Mostafa Taha, Krishna Pabbuleti, Deepak Mane, and Harika Santapuri.

During my dissertation, I had an excellent internship experience at Riscure B.V. I am indebted to Dennis Vermoen, Albert Spruyt, and Niek Timmers for this great opportunity to work with them and learn immensely from them. I feel quite fortunate for meeting and working many brilliant people at Riscure, including Roeland, Nils, Parul, Baris, and Fatih.

I would like to thank the National Science Foundation and Semiconductor Research Corporation for supporting this research through Grant 1441710. I would also like to acknowledge the Bradley Department of Electrical and Computer Engineering, Virginia Tech for supporting me during my research.

I would like to acknowledge Prof. Fatih Ugurdag for being an exemplary mentor and visionary role model to me. Hocam, thank you for your generous support. Without your encouragement and help, I would not be able to start my PhD journey.

I am deeply grateful to Aylin, Harun, Zeliha Teyze, and my little sister Duru for opening their house to me and being like a family to me in Blacksburg. Hoa, it would take another dissertation to describe your positive effect on this research as well as on my personal development. Thank you for enlightening my life and supporting me through these years.

Finally, I owe my deepest gratitude to my beloved family for their unconditional love and support.

Contents

Chapter 1

Introduction

Our daily lives and critical infrastructure are getting increasingly dependent on a spectrum of interconnected embedded systems from low-end smartcards to high-end network equipment. On one end of the spectrum, credit cards process users' bank account information during electronic payments. Smart meters in the electrical power grid provide national-level smart electricity distribution. Electronic passports contain biometric information of millions of citizens. In the middle of the spectrum, applications and services running on the mobile platforms manage sensitive user and corporate data such as credentials for authentication, private information, and identity. On the other end of the spectrum, cloud storage enables users to keep their sensitive data, which may relate to personal or commercial secrets, in third-party data centers. Moreover, modern embedded systems usually store a device-specific firmware, which is a software intellectual property (IP) to configure and control hardware and software components of the system. Considering the increasing dependence on embedded systems to handle sensitive data, those systems must ensure an acceptable level of security during storage, transfer, and use of the sensitive data.

Secure embedded systems employ various mechanisms to satisfy the security requirements (e.g, confidentiality, integrity, authentication) of sensitive data. Encryption algorithms provide confidentiality for keeping information secret from all but autho-

rized parties. Cryptographic hash algorithms enable an entity to check the integrity of data. Message authentication codes and digital signatures are used for verifying integrity and authentication. For instance, the secure boot mechanism ensures secure initialization and configuration of a system by using digital signatures to verify the integrity of the firmware. In addition to these cryptographic mechanisms, the system software of modern embedded devices generally implements access control policies, privilege levels, and isolation mechanisms to manage access to hardware components, files, and software processes.

The aforementioned security mechanisms are subject to attacks from malicious adversaries because of the high value of the protected data. The traditional security threats are logical and mathematical attacks, which are mounted at algorithm or software level. For example, the linear cryptanalysis, differential cryptanalysis, and brute force attacks target the mathematical principles and input/output of cryptosystems. Despite the existence of such attacks, the mathematical principles of the modern cryptographic mechanisms are usually strong enough to withstand those traditional attacks.

Low-level logical attacks tamper with the machine-code level implementation of a software program to subvert the execution of the program and gaining control over the device running the program. In a typical setting (I/O attacker model), an adversary controls the input of a victim program to trigger a memory-safety vulnerability, which allows a program to access a memory location not allocated for that program. For instance, in the infamous buffer overflow vulnerability, an adversary is able to access the memory locations beyond the bounds of an input buffer (i.e, a contiguous chunk of memory locations) due to lack of appropriate checking. As a result, the adversary can gain control over the victim program's execution by in-

jecting malicious code into the memory, corrupting code pointers in the memory, or corrupting security-critical data residing in the memory [1]. Similarly, an adversary can also achieve an out-of-bounds read access and leak confidential information from the memory. In a more advanced setting (memory attacker model), the adversary is also capable of tampering with the virtual memory space of a program to cause confidentiality and integrity problems even if the victim program contains no memory safety vulnerability [2]. Despite their capabilities, the nature and mechanisms of these attacks are well-understood and various effective countermeasures against them have been demonstrated in both hardware and software layer [1, 2].

A relatively new and powerful threat to the security of embedded systems is the implementation attacks. In an implementation attack, an adversary attacks the physical implementation of a security mechanism rather than mathematical or logical vulnerabilities. During the development of previously discussed security mechanisms, software designers assume that the underlying microprocessor hardware always correctly executes instructions of a program, and it is a perfect black box such that adversary cannot observe or manipulate internals of the hardware. However, an embedded system is not a perfect black box in the real world. A software program runs on the physical resources of the microprocessor hardware and interacts with the physical operating environment of the microprocessor through supply voltage fluctuations, heat transfer, operating frequency, radiation emission and so forth. In addition, the assumption of correct hardware operation holds as long as the hardware's operating conditions are within certain margins determined by the implementation of the hardware and the laws of physics. Thus, an adversary can physically observe and alter the execution of a security mechanism through its physical interaction with the operating environment.

The two main categories of implementation attacks are side-channel attacks [3], and fault attacks [4, 5]. In a side-channel attack, the adversary passively observes physical variables (e.g, power consumption) of the security operations, and then correlates these observations with a side-channel leakage model of the secure embedded system. A high correlation reveals information on the sensitive data. In a fault attack, the adversary violates the assumption of correct hardware operation by actively pushing the operating conditions beyond the pre-determined margins. This causes hardware faults. Then the adversary exploits the effects of faults on the execution of the security mechanism and breaks its security. The implementation attacks can break the security of an implementation even if it has no mathematical or logical vulnerability. The pervasive and easy-to-obtain nature of the modern embedded systems makes the impact of implementation attacks more severe.

1.1 Fault Attacks on Embedded Software: Threats and Countermeasures

This research investigates fault attacks on embedded software, which is a powerful hardware-oriented implementation attack against security mechanisms implemented in software running on modern embedded devices. In a fault attack on embedded software, the target of the *fault injection* is the hardware layer while the target of the *fault exploitation* is the software layer. The adversary creates a controlled and engineered fault in the processor hardware, and exploits the effects of the fault on the software to cause sensitive information leakage or to obtain control over the embedded processor. For instance, Figure 1.1 outlines the process of a typical fault attack on unprotected embedded cryptographic software. In this example, the cryptographic

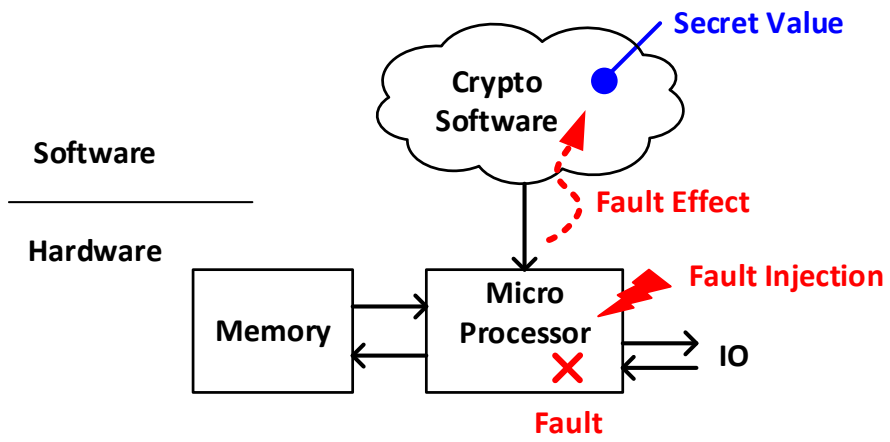


Figure 1.1: In a fault attack on embedded software, faults are injected in hardware, but their effect is exploited in terms of the target software.

software makes use of a secret key, and the adversary aims to learn the value of this key. This requires inducing a specific fault effect in an observable control or data dependency of the software secret. The adversary starts with performing fault injection on the hardware, including the microprocessor pipeline, registers, and memory. Fault injection can be achieved by various means such as reducing supply voltage, increasing operating frequency, or increasing ambient temperature of the processor beyond the allowed margins [6, 7, 8]. In addition, the processor can be exposed to electromagnetic pulses [9] or laser shots [10]. The hardware and the software are connected through the processor's instruction-set architecture, so that a hardware fault in the microprocessor hardware eventually appears in the software as a faulty instruction or as a faulty data value. A faulty instruction changes the meaning of the program, while a faulty data value changes the correctness of the computation. The instruction-faults or data-faults that occur during execution of the cryptographic software are the starting point of a fault exploitation, which eventually enables the adversary to extract the cryptographic key.

Fault attacks initially emerged as theoretical cryptanalytic attacks on cryptographic algorithms [5]. Over the last 20 years, several practical implementations of fault attacks have been demonstrated on both educational evaluation boards [11, 12, 13, 14, 15, 16, 17, 18, 19, 20] and on-the-shelf commercial devices [21, 22, 23, 24, 25, 26, 27, 28] using various fault injection means such as clock glitching [14, 29], voltage starving [7, 30], voltage glitching [18], electromagnetic pulses [31, 32, 33], and laser pulses [10, 34]. In addition to their extensive use in the literature as a cryptanalysis tool on both symmetric and asymmetric cryptography [4], fault attacks have also been shown an effective hacking tool against non-cryptographic, logical security mechanisms such as secure boot, privilege levels, and isolation mechanisms [18, 20, 22, 24, 35, 36]. Implementing a fault attack requires to deliberately control the physical parameters of the operating environment of the target embedded systems, which usually requires physical proximity to the target. However, it has been also demonstrated that even remotely-implemented fault attacks are possible because of the increased complexity of the modern systems [22, 23]. Furthermore, it is also possible to use fault attacks in a combined settings with the logical [37, 38, 39, 40] and side-channel analysis attacks [41, 42, 43, 44].

As fault attacks pose a serious threat to a variety of secure embedded software, there is a rich body of techniques in the literature for fault detection and response. The existing techniques rely on redundancy, which is implemented in either completely in hardware layer or software layer. The first approach (Figure 1.2a) implements both fault detection and response as redundant software design. This approach employs algorithm-specific redundancy, temporal redundancy, or information redundancy [45, 46, 47, 48, 49] at instruction level [49, 50, 51] or algorithm level [52, 53]. This approach verifies the consistency among redundant software executions at var-

ious moments during the operation of the security mechanism. Then it applies an appropriate fault response based on the result of the verification. Based on the used redundancy type, the response can be activating a reset procedure, killing the device, correcting the result, or randomizing the result. The second approach is to implement detection and response at the hardware layer. In this case, one option is the fault-tolerant hardware design, which uses expensive hardware redundancy [54]. A cheaper option is to utilize built-in sensors to monitor low-level operating parameters. This includes current sensors[55], voltage sensors[56], radiation sensors [57] or timing sensors [58]. Fault detectors can also be integrated at the hardware level in the processor such as when using canary registers [59] or ECC memory. When a non-correctable fault is detected in hardware, an immediate hardware-level reaction will be triggered, which may include halting the processor, resetting the processor, or restarting the instruction [59]. The next section discusses the problems of the existing fault attacks and countermeasures, which are still needed to be addressed despite the research effort described above.

1.2 Review of Existing Attacks and Countermeasures

This section reviews the existing fault attacks and countermeasures in separate subsections.

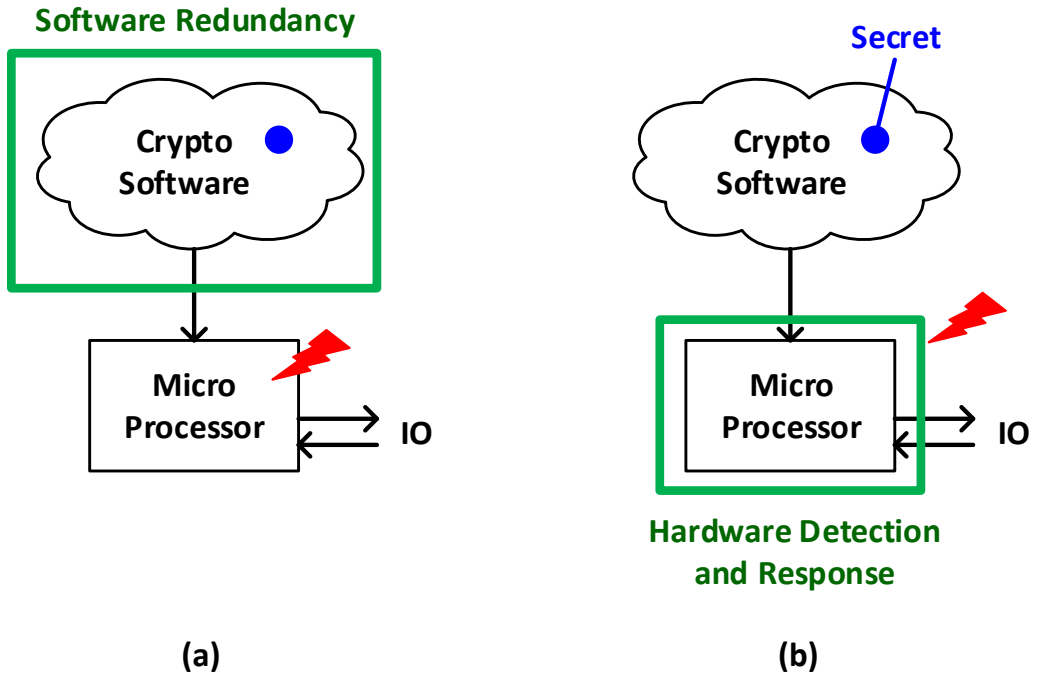


Figure 1.2: Existing strategies for fault countermeasures: (a) Software Detection and Response, (b) Hardware Detection and Response.

1.2.1 Review of the Existing Fault Attacks

In a fault attack, the adversary first designs a fault exploitation method based on a specific fault model, a high-level assumption on the actual induced fault in the software after physical fault injection. Then the adversary manipulates the operating conditions of the microprocessor hardware to induce the assumed fault effect. Although powerful fault exploitation techniques have been developed, their practical implementation is complicated because of the uncertainty that comes with the fault injection process. First, the intended fault effect may not match the actual fault

effect obtained after fault injection. Second, the logic target of the fault attack, the embedded software, is above the abstraction level of physical faults. The uncertainty with respect to the fault effects in the software may degrade the efficiency of the fault attack, resulting in many more trial fault injections than the amount predicted by the theoretical fault attack.

In the literature, the most common strategy to select fault injection parameters is by trial and error, although, in this approach, only a very little portion of the fault injections induce the intended fault effects [18, 19]. This low success rate quickly decreases the efficiency of fault exploitation. For example, Differential Fault Analysis (DFA) technique can recover 128-bit AES key after a single fault perfectly matching the fault model [60, 61]. However, even a mismatch of one in one hundred fault injections can render this technique ineffective [62].

The main reason behind using such an inefficient search strategy is the lack of fault models that accurately capture the fault effects experienced by the software and guide the fault injection process to induce desired fault effects in software. The bulk of the literature on fault attacks treat the modeling of fault behavior of a software program only at the most basic level [4, 8, 63, 64, 65] with a hardware-oriented approach. In this approach, a fault injection method is applied to the microprocessor hardware, and the observed fault effects are captured into a fault model. Typical fault models only capture basic parameters, such as the fault duration (transient or permanent), the fault value (bit-flip, random, stuck-at), and the fault location (a variable, a conditional flag, control flow) [65]. Although it is useful to evaluate the feasibility of fault injection on a processor, such a generic fault model is not sufficient to guide fault injection for inducing the desired fault effects in the target software.

In recent years, this model was further refined by acknowledging a link between fault

injection intensity and the severity of fault effects on the microprocessor [13, 14, 29]. Balasch et al. discuss the case of an AVR micro-controller [29], Moro et al. describe Electromagnetic (EM) attacks on an ARM Cortex-M3 [13], and Korak et al. study the ATxMega-256 as well as the ARM Cortex-M0 [14]. All of them use techniques that trigger timing errors through clock glitches and/or (voltage or EM) pulses. These papers reveal that a gradual increase in fault intensity causes a gradual increase in the number of faults. For example, Balasch shows that opcodes can be gradually changed into NOP (all-zero) from any given opcode, by gradually decreasing the clock period during the instruction fetch. Although this modeling approach enables an adversary to relate the fault intensity to the number of induced faults in the hardware, it is still insufficient in many practical situations, including the case of the embedded software fault attacks examined in this research. There is a need to expand the fault modeling mechanism beyond the basic parameters of fault duration, fault type and fault location. A fault model for embedded software should capture the fault effects experienced by microprocessor instructions, in order to analyze the effects of a fault injection on software and to guide the fault injection. Thus we need to model fault effects in the instruction-set architecture, rather than faults purely at the hardware level.

1.2.2 Review of the Existing Fault Countermeasures

There are several issues with the existing fault attack countermeasures. While redundancy is a generic countermeasure that can capture a wide range of faults, it comes with an inherent performance penalty over the non-redundant design [45, 46, 47]. For embedded implementations, this overhead can be significant. Simple instruction duplication, for example, doubles the execution time regardless of the presence of

a fault [66]. Furthermore, redundancy is no guarantee against fault attacks by an adaptive adversary who targets the consistency checks, or against an adversary with multiple-fault injection capability [17, 19, 20, 67, 68].

From the fault response angle, it is not possible to customize the response to specific requirements of an application when the response is fully implemented in hardware. In this case, the reaction is limited to a generic response. While processor reset may be a viable option in the specialized environment of a smart-card, it may be unacceptable in a complex, multi-purpose environment such as in a system-on-chip. It may also be necessary to differentiate intentional faults from random faults. While the intentional faults affect the sensitive control and data dependencies of the application, random faults affect random parts of an application at random intervals. Therefore, a fixed and uniform fault response may not be desirable for intentional faults [69].

Finally, defending software against fault attacks introduces the additional difficulty that the faults do not originate in the software, but rather in the underlying processor hardware. Therefore, modern embedded systems need *low-cost* and *flexible* mechanisms over multiple abstraction layers for fault detection and response [70, 71, 72, 73].

1.3 Thesis Statement and Research Questions

Multiple abstraction layers (i.e, hardware, instruction-set architecture, software) take part in a fault attack on embedded software. The previous research efforts focus on only a single layer, and this isolated approach causes inefficiency in designing fault attacks and countermeasures.

The aim of this research is to consider the impacts of hardware, instruction-set architecture (ISA), and software layers together to significantly improve the efficiency of attack and protection techniques. This combined approach also yields a deeper, better understanding of fault attack risk on the embedded software. The main research question is as follows:

- How can the knowledge of hardware, ISA, and software layers be combined together to design more efficient fault attacks and countermeasures?

This question is refined into the following sub-questions:

- How should the fault model of a microprocessor be constructed such that it explains the fault injection's effects on the software and guides the fault injection to obtain the desired fault effects?

- Using the fault model, can novel fault attacks on embedded software be designed such that an adversary needs to put less fault injection effort to break the security?

- How can an adversary's view of software execution be reflected into a simulation environment such that software designers can evaluate the fault-attack resistance of their designs at design time?

- How should fault handling be distributed over multiple abstraction layers to provide architectural support for fault-attack resistance satisfying performance and security requirements of modern embedded systems?

1.4 Contributions

This research has four main contributions that are summarized in this section.

1.4.1 Instruction Fault Sensitivity Model

The first contribution of this research is the *instruction fault sensitivity model* that systematically captures fault behavior of a software program running on a microprocessor datapath [11, 12]. An adversary can use the instruction fault sensitivity model to obtain insight into most likely fault effects as well as to pinpoint most sensitive points during the execution of the target software program.

In contrary to the previous fault models, instruction fault sensitivity model characterizes the fault effects at instruction set architecture (ISA) level. This provides several advantages.

- First, the instruction set architecture (ISA) provides an execution model, which allows modeling the impact of faults on the execution of instructions. A typical instruction-execution cycle includes at least instruction-fetch, decode and execute steps. The effect of a fault can change according to each phase of the instruction-execution cycle. For example, a fault on the instruction-fetch could affect the instruction opcode, while a fault on the execution phase could change the instruction operands. Each of these faults has a different effect on the software program.

- Second, the ISA makes a clear distinction between data processing (eg. arithmetic instructions), control (eg. branch instructions), storage (eg. load/store of

data), and input/output operations [63]. Faults have a different effect on each different instruction, and hence software fault models should be instruction-dependent.

- Finally, the data dependencies, which are crucial to understanding the propagation of faults in software, only become visible in the software. It is impossible to analyze fault propagation through the microprocessor hardware alone.

.

To demonstrate the instruction fault sensitivity model, we characterize the fault behavior of a set of SPARC ISA instructions on a 32-bit, 7-stage-pipelined RISC processor. We build the instruction fault sensitivity model against setup-time violation attacks by using gate-level timing simulation. The resulting fault model contains the fault sensitivity value of each (*instruction*, *pipeline stage*) pair. To our knowledge, this is the first ISA-level fault model demonstrated in the literature.

1.4.2 Micro-architecture Aware Fault Injection Attack (MAFIA)

Relying on the instruction fault sensitivity model, we also propose a systematic fault attack methodology, so-called Micro-architecture Aware Fault Injection Attack (MAFIA) [11, 12]. MAFIA enables an adversary to launch an efficient fault attack on an embedded software by exploiting different layers of abstraction. The adversary starts with algorithm-level analysis to determine application-specific fault injection and analysis objectives. Then the adversary studies the software implementation of the algorithm in the instruction level and finds the candidate (*instruction*, *pipeline stage*) pairs for fault injection. Finally, the adversary examines the execution of the

instructions on the pipeline to determine clock cycles to inject faults as well as the fault injection parameters to create the desired effects.

We demonstrate the efficiency of MAFIA with two case studies on a 32-bit RISC processor. First, we develop a Differential Fault Intensity Analysis (DFIA) attack on an unprotected AES software program, and show that the use of the proposed methodology reduces the number of required fault injections an order of magnitude in comparison to the traditional attack methodology. This is the first DFIA attack on a software implementation.

Second, we show how the instruction fault sensitivity model helps to pinpoint the weaknesses of a class of fault attack countermeasures in software, which rely on instruction-level redundancy. We analyze the state-of-the-art instruction-level countermeasures by considering the micro-architectural aspects and identify their vulnerabilities. The analyzed countermeasures include instruction duplication, instruction triplication, instruction-level parity checking, and fault-tolerant instruction sequences. This analysis shows that all of these countermeasures can be broken with a single fault injection. To our knowledge, this is the first work that provides such an analysis of the software countermeasures. We also experimentally demonstrate the feasibility of the aforementioned findings by breaking all of the analyzed instruction-level countermeasures with a single clock glitch injection. This is the first work to demonstrate a practical single-fault attack on instruction-level countermeasures with low-cost clock glitch injection. Further details of MAFIA is explained in Chapter 5.

1.4.3 Micro-architectural Embedded System Simulator (MESS)

Another contribution of this research is a fault attack simulator to analyze the effects of fault injection on the secure embedded software applications, and to assess their vulnerability against fault attacks. We name the proposed simulator as MESS (Microarchitectural Embedded System Simulator). MESS is a micro-architecture level fault injection simulator that considers software-level, ISA-level, and hardware-level components of fault attacks together. Therefore, MESS is capable of reflecting the attackers' view of execution into the simulation environment. This allows a user of MESS to simulate various real-world fault attack scenarios.

MESS potentially enables software designers to evaluate the security of their designs at the early phases of the development, which is crucial for efficient cost-performance-security trade-offs. Another potential use of MESS is to security evaluators to discover novel threat models, to explore architecture-specific fault models, to carry out a root-cause analysis, and to guide the actual fault injection.

We implement MESS by extending a widely used, cycle-accurate, micro-architecture level, full-system simulator gem5 [74] with fault injection and analysis capabilities. gem5 enables a user to define main micro-architectural components of a processor such as pipelined datapath, caches, and memories. Then the user can run an application on this model and analyze its execution. The current implementation of MESS supports x86 and ARM instruction set architectures. Chapter 6 explains the details of MESS.

1.4.4 Fault-attack Aware Microprocessor Extensions (FAME)

To mitigate the fault attacks, we propose Fault-attack Aware Microprocessor Extensions (FAME) that transform an unprotected baseline processor to a fault-attack-resistant processor. The main objective of FAME is to provide low-cost architectural support for fault mitigation in the processor hardware to enable software designers to develop cheaper and more secure fault attack countermeasures. FAME achieves its objective by distributing fault detection and response across multiple abstraction layers.

FAME combines the strong points of the previous techniques. It consists of fault detection in hardware, followed by hardware-level checkpointing, and a secure trap in software (Figure 1.3). FAME instruments the microprocessor with sensors to pick up fault injection attempts. The microprocessor maintains a hardware checkpoint of low-level critical system state that is updated every clock cycle. When the sensors detect a fault injection, the checkpoint is frozen and the fault detection hardware initiates a secure trap handler in software. The secure trap handler recovers the critical system state from the checkpoint and restores it. The secure trap handler then executes a user-defined fault-response to implement application-specific countermeasures. FAME does not suffer from the performance overhead of redundant software execution, and it responds to a fault injection before its effect can spread.

Compared to software redundancy and software checkpointing, a hardware sensor offers low response latency, and it may enable detection of a fault injection attempt before an actual fault in the processor occurs. The use of a sensor brings a significant advantage, as it also reduces the cost of the overhead. For embedded or performance-sensitive implementations, minimal performance overhead is an important concern.

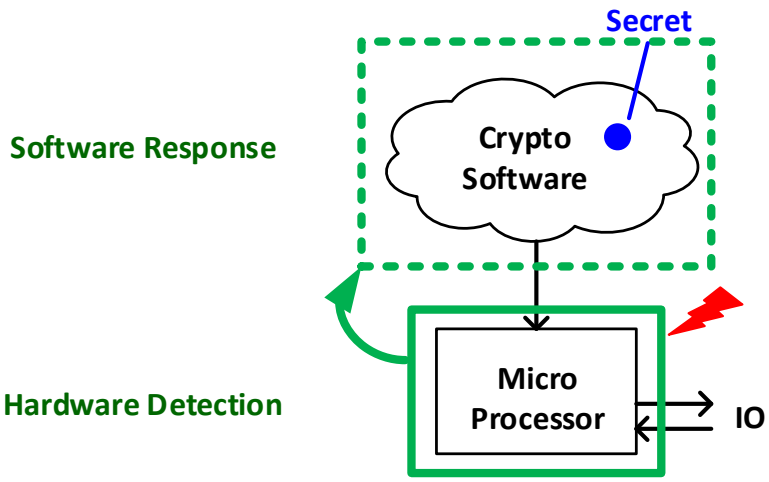


Figure 1.3: Key principle of the proposed fault-attack-resistant FAME processor: Hardware Fault Detection, Software Fault Response.

However, a sensor is subject to false-negative or false-positive alarms. That is, not every fault injection may trigger the sensor, and not every detected fault is malicious. We address this challenge by designing our chip with a specific attacker model in mind. That is, we argue that a countermeasure has to be designed with the attack mechanism in mind, in order to justify the cost and overhead relative to the gain in security.

We designed a prototype of FAME and implemented it both on an FPGA [75] and as a chip. The prototype contains a RISC processor with fully integrated fault-attack-resistant extensions and the capability to execute a secure trap. It also contains a detailed fault-injection and debugging infrastructure to assist the development and testing of fault attacks on the chip. The results of the prototype demonstrate that FAME is a generic, low-cost, performance-efficient, flexible, and backward-compatible countermeasure of fault attacks. This is the first work that investigates architectural

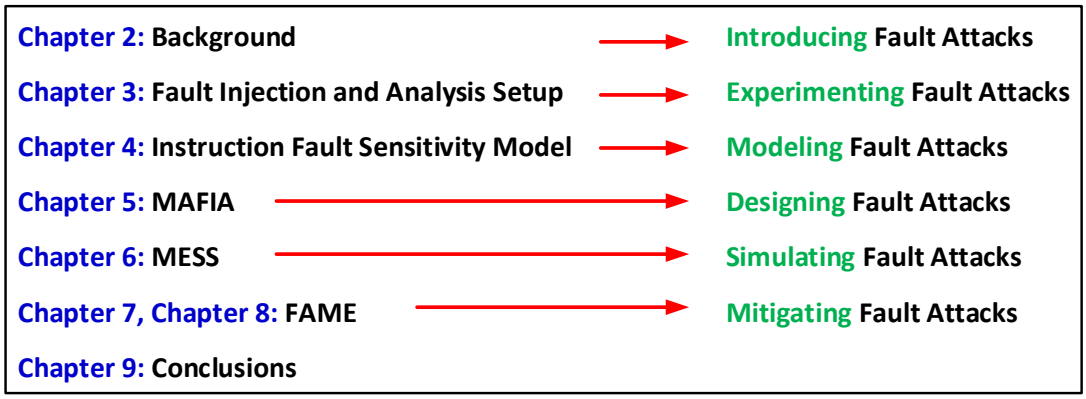


Figure 1.4: Roadmap for the dissertation.

support to assist software developers in developing better countermeasures. Chapter 7 and Chapter 8 give the details of FAME.

1.5 Organization of the Dissertation

The remaining part of the dissertation is organized as follows (Figure 1.4): Chapter 2 explains the basics of fault attacks and the steps that an adversary has to go through to build a fault attack. Chapter 3 explains the details of the fault injection analysis environment used to experimentally verify the results. Chapter 4 presents the instruction fault sensitivity model. Chapter 5 explains the steps of MAFIA and demonstrates its efficiency with two case studies. Chapter 6 provides details of MESS, and case studies to demonstrate its use. Chapter 7 details FAME and its chip prototype. Chapter 8 presents experimental evaluation results of FAME. Finally, Chapter 9 concludes the dissertation.

Chapter 2

Background

This chapter provides background information about fault attacks on embedded software.

2.1 Threat Model

The aim of a fault attack is breaching the security of a software program by forcing a security-sensitive asset into unintended behavior. For this purpose, the adversary injects well-crafted, targeted hardware faults by deliberately altering the operating conditions of the microprocessor that runs the target software. Then the adversary exploits the effects of the faults on the target software's execution and breaks the security. Consequently, the target of exploitation is the software layer while the origin of vulnerability (i.e, faults) is the hardware layer.

In a typical fault attack, the adversary is not capable of directly modifying or monitoring the internals of a chip, or changing the binary of a program. The adversary is able to alter the execution of a target program by controlling the physical operating conditions (e.g, timing, supply voltage, temperature) of the processor hardware executing the program. The adversary can also provide input to the target program, and observe the effects of abnormal operating conditions on the software execution

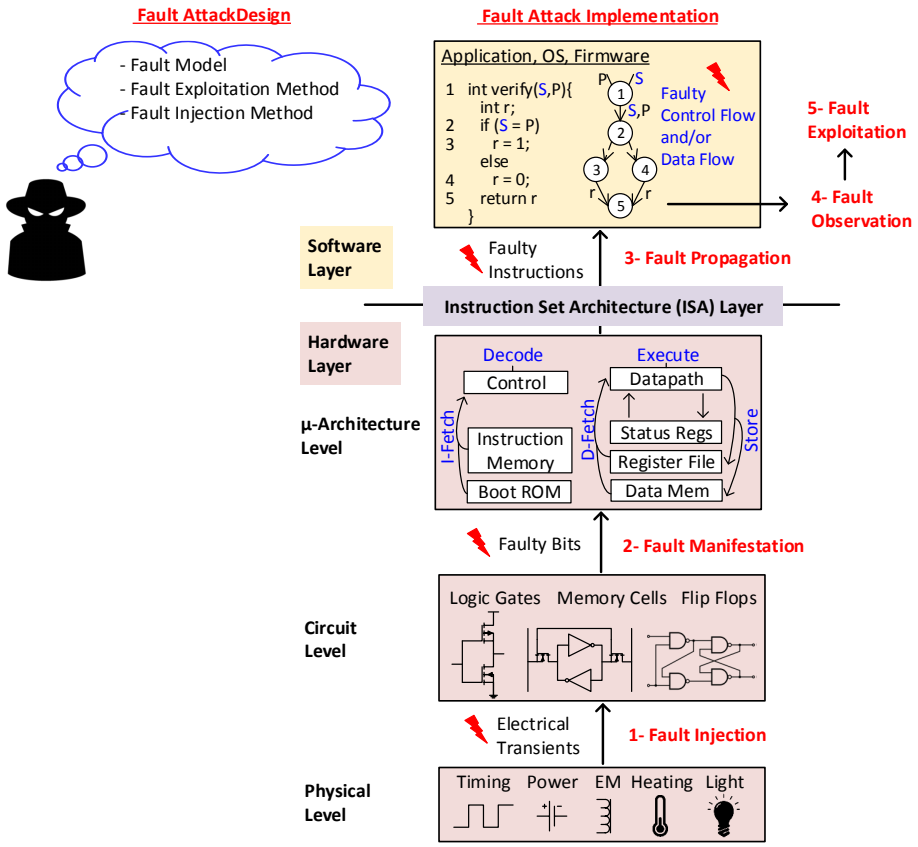


Figure 2.1: Anatomy of a typical fault attack on embedded software: The target of fault injection is the hardware while the target of exploitation is the software.

through system output or a related side-channel such as power consumption, cache-activity-related timing, and performance counters.

2.2 Using Faults as a Hacking Tool

Figure 2.1 illustrates the steps and mechanisms involved in a typical fault attack on embedded software. A fault attack consists of two main phases, fault attack design

and fault attack implementation (Steps 1-5 in Fig. 2.1). In the design step, the adversary analyzes the target to determine fault model (i.e, an assumption on the faults to be injected), fault exploitation method, and fault injection technique. For instance, an adversary may intend to inject faults into several assets such as an encryption program, a security-related verification code, a memory transfer function, the processor state register, a system call, the firmware, or configuration information of the target device. The adversary may then exploit the fault effects on the target asset for various attack objectives such as weakening the security, bypassing security checks, intellectual property theft, extracting the confidential data, privilege escalation, activating debug modes, and disabling secure boot of the device.

The implementation phase is a combination of five steps:

1. **Fault Injection:** In this step, the adversary applies a physical stress on the microprocessor to alter its physical operating conditions and to induce hardware faults. The applied physical stress can be in various forms such as clock glitches, supply voltage glitches, electromagnetic (EM) pulses, and laser shots.

   To induce the desired faults, the adversary varies *fault intensity*, which is the degree of the physical stress by which the microprocessor hardware is pushed beyond its nominal operating conditions. The adversary controls the fault intensity via fault injection parameters. For clock glitching, shortening the length of the glitch increases the fault intensity. It is controlled by glitch/pulse voltage and length for voltage glitching, electromagnetic pulse injection, and laser pulse injection. The laser and electromagnetic pulse injections also enable the adversary to localize the fault intensity by controlling the shape, size, and position of the injection probe.

2. **Fault Manifestation**: The circuit-level effect of fault injection is creating electrical transients on the nets, combinational gates, flip-flops, or memory cells. A fault manifests at the micro-architecture level when the electrical transients are captured into a memory cell or flip-flop, and change its value.

   The number of manifested faulty bits in the micro-architecture level is correlated to the applied fault intensity: A gradual change in the fault intensity causes a gradual change in the manifested faults. We call this relation *biased fault behavior*. This behavior is valid independent of the used fault injection method, and it enables the adversary to control the induced fault effects [12, 13, 14, 76]. Because of the biased fault behavior, the adversary is able to find a critical fault intensity value, at which the electrical transients become strong enough to cause fault manifestation. That critical fault intensity value is called *fault sensitivity* of the target hardware [77].

3. **Fault Propagation**: In this step, the effects of the manifested faults are propagated to the software layer through execution of faulty instructions. The next two paragraphs briefly explain the mechanism behind fault propagation.

   Software security mechanisms are implemented as a sequence of instructions executed by the microprocessor hardware. In addition, each instruction goes through the *instruction-execution cycle* that consists of multiple steps carried out by a certain subset of available micro-architecture-level hardware blocks. The processor loads each instruction from program memory (*instruction-fetch*), then determines the meaning of the current instruction through its opcode (*instruction-decode*), then executes the current instruction (*instruction-execution*), and then updates the state of the processor based on the instruction's result (*instruction-store*). The number of steps in the instruction-execution cycle is

architecture dependent, and it can vary considerably from one microprocessor to the next.

The manifested faults may cause faulty bits in any micro-architectural hardware block such as instruction memory, controller, datapath and register file. The effects of the manifested faults are propagated to the software layer when an instruction uses the affected micro-architectural block. As each instruction uses a specific subset of the micro-architectural blocks, the precise effect of a hardware fault depends on the type of the instruction. For instance, a bit-flip fault injected during the execution step of an addition instruction may yield a single-bit fault in the result of this instruction. However, the same bit-flip fault injected during a memory-load instruction would cause a single-bit fault in the effective address calculation, and thus, data is loaded from a wrong memory location. In the former case, only a single bit of the destination register is faulty; while in the latter case the destination register has a random number of faulty bits.

4. **Fault Observation:** An adversary needs to observe the effects faulty instructions in order to exploit them. An observable fault effect can be a faulty system output such as a faulty ciphertext, a side-channel information such as a sudden change in the power consumption, a single-bit information showing if fault injection was successful, or micro-architectural affects observed through performance counters [78, 79]. These effects become observable to the adversary when they are subsequent instructions that have data-dependencies or control-dependencies on the faulty instruction are executed.

5. **Fault Exploitation:** In the final step, the adversary exploits the observable fault effects and breaks the security. For example, the adversary can analyze

the differential of the correct and faulty ciphertexts from a cipher to retrieve the secret key used for the encryption. For the same purpose, an adversary may also use a single-bit side-channel information of whether fault injection was successful. Similarly, the adversary may use the faults to trigger traditional logical attacks such as buffer overflows and privilege escalation.

2.3 Fault Injection Techniques

In a fault attack, it is essential to induce well-controlled faults during execution of the target software. An adversary achieves fault injection by deliberately applying physical stress to push the operating conditions of the underlying microprocessor hardware beyond their allowed margins. The adversary controls the induced faults through timing, location, and intensity of fault injection. The *timing of fault injection* is defined as the moment at which physical stress is applied to the processor. The *location of the fault injection* is the spatial portion of the processor that is exposed to physical stress. The *intensity of the fault injection* is the amount of physical stress applied to the processor.

This section discusses common techniques used for fault injection. We briefly describe main characteristics of each fault injection technique. We partition the fault injection techniques into two main categories (Figure 2.2): Hardware-controlled fault injection and software-controlled fault injection.

Hardware-controlled fault injection techniques employ a separate external fault injection hardware to apply physical stress to the target hardware and induce faults in the victim software. Typically, the fault injection process is controlled by an-

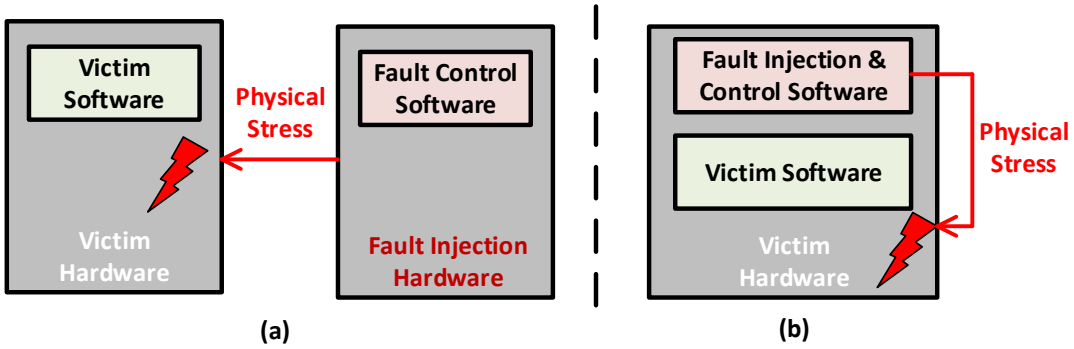


Figure 2.2: Fault Injection Categories: (a) Hardware-Controlled Fault Injection (b) Software-Controlled Fault Injection

other software program (i.e, fault control software) running on the fault injection hardware. *In software controlled fault injection techniques*, fault injection is controlled with a malicious software (i.e, fault injection and control software), which runs on the same hardware platform as the target software does. This malicious software alters the physical operating conditions of the target hardware to induce faults. While hardware-controlled techniques require physical proximity to the target system, software-controlled fault injection techniques enable remote fault attacks.

2.3.1 Hardware-controlled Fault Injection Techniques

Several hardware-controlled fault injection techniques have been successfully demonstrated in the literature [8, 30, 78]. The following sections provide an example list of commonly used techniques.

Tampering with Clock Pin

An adversary may inject faults by tampering with the external clock signal of the target device.

One way of exploiting the clock signal for fault injection is *overclocking* [80], in which the adversary persistently applies a higher-frequency clock signal than the nominal clock frequency of the device. This violates setup-time constraints of the device and causes premature latching of the faulty values in flip-flops of the device [81]. The spatial precision of this method is low because the modifications in the external clock signal are distributed across the whole chip surface through a clock network. Similarly, the temporal precision of overclocking is also low because all of the clock cycles are affected by fault injection; the adversary cannot select the clock cycles to be affected by the fault injection. On the other hand, the adversary has a fine control on the fault intensity through clock frequency.

Another way of tampering with the clock signal is *clock glitching* [14], in which the adversary temporarily shortens the length of a single clock cycle. This causes setup-time violations during the affected clock cycle. In comparison to overclocking, the adversary has a precise control on the temporal location (i.e, timing) of the fault injection. The intensity of the fault injection is controlled through the length of the glitched clock cycle. Similar to the overclocking, the spatial precision of clock glitching is low.

For the clock glitching and overclocking techniques, the state-of-the-art fault injection setups [16, 82] provide nanosecond-level temporal precision. The disadvantage of tampering with the clock signal is that this method requires physical access to an external clock pin. If a device uses an internally-generated clock signal, using this

method is infeasible.

Tampering with Supply Voltage Pin

An adversary can also inject faults by altering the external supply voltage of the target device. The adversary may use *underfeeding* [7], in which a lower voltage than the nominal voltage is supplied to the device. Lower supply voltage increases the delay of combinational paths. This causes setup-time violation when the voltage drop is large enough to make a path delay larger than the applied clock period. This method has low spatial precision as the supply voltage is distributed all over the chip through a power network. Similarly, the temporal precision of the fault injection is low because all of the clock cycles are exposed to the lower supply voltage. The adversary controls the fault intensity through the value of the external supply voltage.

The adversary can also use *voltage glitching* [18], which injects temporary voltage drops and provides the capability to control the temporal location of the fault injection. In this case, the adversary controls the intensity with the glitch offset from the sampling edge of the clock signal, glitch voltage, and glitch width similar to the clock glitching.

These methods require physical access to the supply voltage pin. Removing the external coupling capacitance on the supply voltage line improves the efficiency [18]. The drawback of tampering with external voltage pin is that the adversary does not have precise control on the timing and location of the fault injection.

Tampering with Operating Temperature

An adversary may also use *overheating* to trigger setup-time violations [81, 83] for fault injection. In this method, the adversary does not have precise control on the spatial and temporal location of the fault injection. The intensity of fault injection is controlled via operating temperature of the target device.

In addition to the setup-time violation on the datapath, overheating also causes modification in memory cells in EEPROM [84], Flash [84], and DRAM [85] memories. While Govindavajhala et al. [85] use a low-spatial-precision light bulb as the heating source, Skorobogatov [84] employs a 650nm-wavelength laser to increase the spatial precision of heating.

Combination of Voltage, Frequency, and Temperature Fault Injection

Zussa et al. [81] demonstrated that overclocking, clock glitching, voltage glitching, underfeeding, and overheating exploit the same fault injection mechanism, which is the violation of a device's setup-time constraints. In addition, Korak et al. [14, 86] showed overheating and voltage glitching improves the efficiency of clock glitching.

Optical Fault Injection

In the optical fault injection, the adversary decapsulates the target integrated circuit (IC) and exposes the silicon die to a light pulse. The applied light pulse induces a photo-electric current in the exposed area of the IC, which then cause faulty computations [87]. The spatial location is controlled by the position and the size of the light source, and the temporal location is controlled by the offset of the pulse from

a trigger signal. The intensity of the fault injection is determined by the energy and duration of the light pulse. It has been demonstrated that optical fault injection can be achieved with a low-cost camera flash light [34, 87]. The state-of-the-art optical fault injection setups [10] use laser beams for fault injection to achieve micrometer-level spatial and nanosecond-level temporal precision. They also provide precise control on the fault intensity. This enables an adversary to target a single transistor. A disadvantage of the optical fault injection is that it requires decapsulation of the target IC. In addition, it can permanently damage the target IC.

Electromagnetic Fault Injection

In electromagnetic fault injection (EMFI), the adversary applies transient or harmonic EM pulses on the target integrated circuit (IC) through a fault injection probe, which is designed as an electromagnetic coil. The adversary places the probe above the target IC and applies a voltage pulse to the coil, which induces eddy currents inside the target IC. Then the effects of the induced eddy currents are captured as faults. The adversary controls the temporal location of fault injection through offset of the EM pulse from a trigger signal. The spatial location of the fault injection is controlled via position and size of the injection probe. The fault intensity is determined by the voltage and duration of the applied EM pulse. The feasibility of EMFI on off-the-shelf microprocessor ICs has been demonstrated using both low-cost and high-cost injection setups. For instance, Schmidt et al. [34] use a simple gas lighter to induce EM pulses onto an 8-bit microcontroller with low spatial and temporal precision. The state-of-the-art EMFI setups [31, 88, 89] provide millimeter-level precision in spatial location and nanosecond-level precision in the temporal location of the EM pulse. Furthermore, these setups also provide precise control on the voltage

and duration of the applied EM pulse. The advantages of EMFI is that it does not require decapsulation of the target IC and it can inject local faults. However, its spatial precision is lower than the spatial precision of the laser fault injection.

2.3.2 Software-controlled Fault Injection Techniques

Software-controlled fault injection is a recently discovered research area. The following two sections briefly explain the existing two software-controlled fault injection technique.

Tampering with DVFS Interface

In the modern embedded systems, Dynamic Voltage Frequency Scaling is a commonly used energy management technique, which regulates the operating voltage and frequency of a microprocessor based on its dynamic workload. In a typical DVFS scheme, kernel-level drivers control the frequency and voltage of a processor through on-chip regulators. Tang et al. [22] demonstrated that an adversary can induce faults by exploiting the interface between the software drivers and hardware regulators. In this technique, the adversary uses a malicious kernel-level driver to push the operating voltage and/or frequency of the processor beyond the allowed margins. This violates the setup time constraints and causes faults. This method allows an adversary to overclock and/or to underfeed the target device for a specific period of time. The adversary controls the temporal location with the endpoints of the overclocking or underfeeding period. As both the clock and voltage signals are chip-level global signals, the adversary does not have a direct control on the spatial location. The intensity of fault is determined by the overclocking frequency and the

underfeeding voltage value. This method requires neither additional fault injection hardware nor physical access to the target device.

Triggering Memory Disturbance Errors

In this fault injection method, the adversary injects faults into memory cells by exploiting the reliability issues of modern memory hardware such as DRAM and Flash memory chips. The continuous scaling down in the process technology has enabled memory manufacturers to significantly reduce cost-per-bit by placing smaller memory cells closer to each other. However, this also increases electrical interference between memory cells: Accessing a memory cell electrically disturbs nearby memory cells. A disturbed memory cell loses its value and experiences a memory disturbance error when the amount of electrical disturbance is beyond noise margins of that disturbed cell [90, 91].

An adversary may trigger memory disturbance errors through a non-privileged fault injection program. This program repeatedly accesses a set of memory cells (i.e, aggressor memory cells) to induce disturbance errors in a set of victim memory cells storing security-sensitive data. This method allows an adversary to corrupt memory space of a security-sensitive program from memory space of the adversary-controlled fault injector program. Memory disturbance errors have been demonstrated on commodity DRAM and NAND Flash memory chips [90].

In DRAM memories, the memory disturbance errors are induced through Rowhammer mechanism [91]. Thus, it is called Rowhammering. A DRAM memory is internally organized as a two-dimensional array of DRAM cells, where each cell consists of an access transistor and a capacitor storing charge to represent a binary value.

As capacitors lose their charges because of the leakage current, the DRAM cells are periodically refreshed to restore their charges. Each row of the array has a separate wordline, which is a wire connecting all memory cells on the corresponding row. To access a DRAM cell within the two-dimensional array, the corresponding row of the array is activated by raising the voltage of its wordline. Persistent access to the same row causes repeated voltage fluctuations on its wordline, which electrically disturbs nearby rows. This disturbance increases the charge leakage rate in the nearby DRAM rows [91]. As a result, a memory cell within a nearby row experiences a memory disturbance error (a bit-flip error) if it loses a significant amount of charge before it is refreshed. An adversary may take advantage of that physical phenomenon to inject faults. For this purpose, the adversary runs a malicious fault injection program on the target processor, which aims at altering a security-sensitive state of a victim program running on the same processor. The fault injection program continuously accesses an aggressor DRAM row in its own memory space and induces faults into a victim DRAM row within the victim program's memory space [23, 25, 92].

Similar disturbance mechanisms have been also demonstrated on multi-level cell (MLC) NAND Flash memories. Similar to DRAM memory, a Flash memory is also internally organized as an array of Flash memory cells, each of which is a floating-gate transistor. The amount of charge stored in the floating gate determines the threshold voltage of the transistor, which is used to represent the stored data, In MLC Flash memories, each cell stores two bits of data. Unlike DRAM memories, Flash memories do not require periodic refreshing. Cai et al. [90] demonstrated that the capacitive coupling between neighboring Flash cells enables two memory disturbance error mechanisms. The first mechanism, Cell-to-Cell Program Interference (CCI), introduces faults into a Flash cell when a nearby cell is programmed (i.e,

written). The amount of interference is high when a specific data pattern for programming is used. Cai et al. [90] and Kurmus et al. [93] showed how a malicious fault injection program may trigger CCI mechanism to cause a security breach. The second mechanism is *Read-Disturb*, in which the content of a Flash cell is disturbed when a nearby cell is read. Cai et al. [90] demonstrated the use of read-disturb to cause security problems.

The advantage of fault injection by triggering memory disturbance errors is that it can induce single-bit to multi-bit faults into a certain memory location [92]. This enables an adversary to break several security mechanisms.

2.4 Fault Manifestation in the Micro-architecture

This section explains the effects of physical fault injection on the micro-architecture of the target processor. First, we will distinguish micro-architecture (i.e, internal architecture) of a processor from its architecture (i.e, external architecture). Then we will briefly explain main characteristics of the induced faults into micro-architecture.

Any processor can be described from two distinct architectural perspectives. The architecture of the processor describes it as seen by programmers in terms of its instruction set and facilities. The architecture defines semantics and syntax of available instructions, program-visible processor registers, memory model, and how interrupts are handled. It is the boundary between hardware and software as well as a contract between programmers and hardware designers. The micro-architecture describes the physical organization and implementation of the architecture. This includes the memory hierarchy, pipeline structure, available functional units, employed mech-

anisms (e.g, out-of-order execution) for instruction-level parallelism, and so forth. The micro-architecture is optimized to satisfy cost and performance requirements.

Faults manifest as incorrect bits in the flip-flops or memory cells employed in the micro-architecture if the applied physical stress is beyond noise margins of target processor hardware. The parameters and type of physical fault injection technique determine characteristics of the manifested faults. In this work, we use four parameters to describe any manifested fault in micro-architecture level:

- **Location of the Manifested Fault:** This parameter specifies the micro-architectural blocks that contain faulty bits because of physical fault injection.

  Faults may manifest in any micro-architectural block in the control or datapath part of the processor such as instruction memory, instruction fetch block, instruction decode block, operand fetch block, execution block, data memory, register file, processor status register, and conditional flags. An adversary's control on the location of the manifested faults depends on the spatial precision of the used fault injection method, which is characterized as precise control, loose control, and no control [63].

- **Size of the Manifested Fault:** This parameter specifies the number of faulty micro-architecture bits induced by physical fault injection. An adversary can control the size of the manifested faults by adjusting the fault intensity. In the literature, manifested faults are commonly classified as single-bit faults, byte-size faults, word-size faults, and arbitrary-size faults [65].

- **Effect of the Manifested Fault:** This parameter specifies the logical effect of the manifested fault on the fault location. Common fault effects are stuck-at fault, bit-flip fault, bit-set fault, bit-reset fault, and random fault [65].

- **Duration of the Manifested Fault:** Fault attacks exploit provisional faults, of which effect last as long as the physical stress is applied. The faults are recovered when a new value is written into the faulty flip-flop or the memory cell.

The next section explains how the manifested faults propagate to the software layer.

2.5 Fault Propagation to the Software Layer

The manifested faults propagate to the software layer as faulty instructions when the micro-architectural blocks containing the faulty bits are used by the instructions of the target program. Propagated fault effects are determined by the type of affected instruction, type of faulty micro-architectural block, and the characteristics (size, effect) of the manifested faults. As each processor implementation has its own micro-architecture, it is not possible to list all of the potential fault effects propagated to software. Instead, we provide an example to demonstrate a list of potential fault effects for a subset of SPARC instructions running on a hypothetical generic micro-architecture. Using the same approach, similar lists can be built for specific instruction sets and processor implementations.

We chose four SPARCv8 instructions: a memory-load (`ld`), a logic (`xor`), a comparison (`cmp`), and a conditional branch (`be`) instruction. Table 2.1 lists the instructions and their definitions.

The assumed generic micro-architecture contains the following blocks to carry out instruction-execution cycle for each instruction:

Table 2.1: An Example Set of SPARCv8 Instructions

Instruction	Definition
`ld [r1+r2], r3`	Loads a 32-bit word into register `r3` from data memory (D-Mem) address `r1+r2`.
`xor r1, r2, r3`	Bit-wise XOR operation on `r1` and `r2` Result is written to register `r3`.
`cmp r1, r2`	Compares registers `r1` and `r2` and updates conditional flags accordingly.
`be offset`	PC-relative conditional jump: If zero-flag is set, PC will be `PC + offset`. Otherwise, PC will be `PC + 4`.

- **I-Mem Block** is the instruction memory that stores the instructions.
- **I-Fetch Block** prepares the address for the instruction memory, program counter (PC). Then, using the prepared PC, it fetches an instruction into the instruction register (IR).
- **I-Decode Block** takes the fetched instruction from IR and decodes it to determine the location of the source operands, the location of the destination operands, and the operation to be applied. The source operands are fetched from the register file. The destination may be the register file, data memory (D-Mem), or conditional flags.
- **O-Fetch Block** uses the decoded information to fetch the input operands from the register file and to feed them to the execution block. The `be` instruction does not use this block because it does not fetch any operand from the register file.
- **Execute Block** applies the required operation on the fetched source operands and generates a result. For `ld`, it calculates the D-Mem address from `r1` and

Table 2.2: Propagated effects to software layer for each faulty micro-architectural block (with 1-bit fault) and instruction

Faulty Block	**Propagated Fault Effects**			
(1-bit Fault)	`ld [r1+r2], r3`	`xor r1, r2, r3`	`cmp r1, r2`	`be dest`
I-Mem I-Fetch (PC, IR) I-Decode	Execution of a wrong instruction due to opcode-field corruption Fetching operands from wrong location due to source-operand-location corruption Updating a wrong destination due to destination-operand-location corruption Fetching next instruction from a wrong address due to PC corruption			
O-Fetch (1-bit fault in `r1` or `r2`)	Arbitrary # of faults in `r3` (Faulty D-Memory address)	1-bit fault in `r3` (Faulty XOR input(s))	Faulty update of conditional flags	No effect
Execute	Arbitrary # of faults in `r3` (Faulty D-Memory address)	1-bit fault in `r3` (Faulty XOR operation)	Faulty update of conditional flags	1-bit fault in jump address or Inversion of branch
Store	1-bit fault in `r3` (Faulty update of `[r1+r2]`)	1-bit fault in `r3` (Faulty update of `r3`)	Faulty update of conditional flags	1-bit fault in jump address
D-Mem	No effect			
Register File	Fetching wrong source operands from register file			No effect
Conditional Flags	No effect			No jump to `dest`

`r2`. For `xor`, it applies bitwise XOR operation on `r1` and `r2`. For `cmp`, it subtracts `r2` from `r1`. For `be`, it calculates the destination address from the current PC and `offset`. It also checks the conditional flags to determine if the branch will be taken.

- **Store Block** updates the destination location (D-Mem, register file, or flags) with the result computed by the execution block. For the `ld` and `xor`, it is the register `r3`. For the `cmp` instruction, the destination is conditional flags. For the `be` instruction, the destination is the PC value if the branch is taken. Otherwise, it will not affect any destination.

Table 2.2 provides an example list of propagated fault effects for each (*instruction, micro-architecture block*) pair. In this example, we assume a *single bit-flip fault* in any micro-architectural block. A fault induced in I-Memory, I-Fetch, or I-Decode block would affect syntax (i.e, opcode and operands) and/or semantics (i.e, the operation to be applied) of an instruction independent from the type of the instruction. Thus, Table 2.2 shows the propagated fault effects for these blocks in a single cell. Faults

induced in the other blocks would cause errors in the instruction-specific computation of a correctly fetched and decoded instruction:

- **I-Mem, I-Fetch, I-Decode:** If the fault manifests in the opcode part of the faulty instruction, another instruction will be executed. If the fault affects the addresses of source operands, they will be fetched from an incorrect location. Similarly, the result of an instruction will be written into a wrong location if the fault hits the destination address. Finally, the next instruction will be fetched from an incorrect location if the PC calculation gets faulty.

- **O-Fetch:** For the `ld` instruction, a single-bit fault in this block affects the value of register `r1` or `r2` fetched from register file. The fault then causes D-Mem address to be faulty. As a result, a single-bit fault in either `r1` or `r2` may induce an arbitrary number of faults in the destination register `r3` because the result will be fetched from an incorrect D-Mem location.

  For the `xor` instruction, the fault will affect a single bit of `r1` or `r2`, which will be propagated to `r3` as a single-bit fault.

  For the `cmp` instruction, the single-bit fault may affect the result of the comparison, which will alter the conditional flags based on the modified comparison result.

  For the `be` instruction, the fault will not have any effect because this instruction does not fetch anything from the register file.

- **Execute:** For the `ld`, `xor`, and `cmp` instructions, the effects of the fault will be same as the effects described in the O-Fetch case.

  For the `be` instruction, the fault will change the single-bit of the computed

branch address. If the branch is taken, the destination address will be wrong. Otherwise, the faulty branch address will not affect the program. The fault may also change the direction of the branch instruction from taken branch to non-taken branch, or vice versa.

- **Store:** For the `ld` instruction, the fault will cause a single-bit error in the correctly computed D-Mem address `[r1+r2]`. For, the `xor`, and `cmp` instructions, the effects of the fault will be same as the effects described in the O-Fetch case. For the `be` instruction the fault will change the value of the PC if it is a taken branch.

- **D-Mem:** As none of the instructions use a value from the data memory, the fault in D-Mem will not affect any of the considered instructions.

- **Register File:** For the `ld`, `xor`, and `cmp` instructions, the effects of the fault will be same as the effects described in the O-Fetch case. As the `be` instruction does not use this block, the fault will not have any effect on this instruction.

- **Conditional Flags:** The fault in conditional flags will affect only the `be` instruction as the other instructions do not use the conditional flags.

2.6 Fault Exploitation Techniques

This section presents main fault exploitation techniques, which have been proposed to break the security of both cryptographic and non-cryptographic security mechanisms protecting embedded software. Each exploitation technique relies on a fault model, which is a high-level assumption for the effects of physical fault injection on the

execution of the target software. Thus, we start with commonly used fault models in practice. Then we will briefly explain fault exploitation techniques.

2.6.1 Fault Models

In the design phase of a fault attack, an adversary makes a fault model assumption and develops an exploitation strategy based on the fault model. This assumption generally includes the location of the fault in the data or control flow of the target program, the timing of the fault with respect to the duration of the target program, size of the fault, and effect of the fault. The fault models can be described in algorithm level, source code level, or instruction level. The following paragraphs provide an example list of commonly used fault models.

The most of fault-based cryptanalysis techniques on symmetric and asymmetric cryptography assume faults on data flow of a target program that corrupt a single bit, single byte, multiple bytes, or a single word of a security-critical variable in various ways (e.g, flip, set, reset, random) [64, 65, 78].

On the control flow, the most popular fault models are to skip the execution of a specific instruction (i.e, instruction skip) [38, 94], multiple instruction skips [33, 95], replacing an instruction with another one (i.e, instruction modification) [18, 29], changing the result of a conditional branch [40, 96], and tampering with loop counters [97, 98].

In the implementation of a fault attack, the adversary aims at inducing the fault effects assumed in the fault model via fault injection, fault manifestation, and fault propagation processes. Therefore, a fault model can be realized through different combinations of fault injection, fault manifestation, and fault propagation. The

following sections provide a list of commonly used fault exploitation techniques to breach the security of embedded software.

2.6.2 Cryptanalysis using Fault Injection

Using fault injection for cryptanalysis has been extensively studied on the implementations of symmetric-key, public-key, and post-quantum cryptography algorithms [8, 63, 64, 65, 78].

Differential Fault Analysis (DFA) is the most widely used fault-based cryptanalysis technique. The main principle of DFA is to exploit the differential between the faulty and fault-free outputs of a cryptosystem. In a typical DFA attack, an adversary collects two outputs (e.g, ciphertexts) from a cryptosystem (e.g, encryption) that are generated for the same input (e.g, plaintext) and secret variable (e.g, encryption key). One of the outputs is collected without fault injection. During the generation of the second output, the adversary injects a certain fault into the execution of the cryptosystem. Then the adversary analyzes the propagation of this fault differential to the output and reveals the secret variable. DFA attacks assume specific fault during differential analysis of the faulty and fault-free outputs. Various DFA techniques have been successfully demonstrated on block ciphers [99], stream ciphers [100], public-key algorithms [78, 101], and post-quantum cryptography [102].

Safe Error Analysis (SEA) attacks exploit the dependence between the use of a faulty data and the value of a secret variable. An adversary first identifies a target intermediate variable, of which use depends on the value of a secret variable. Then

the adversary injects a specific fault into the target variable and observes whether the output is faulty or not. If the output is faulty, it means that the faulty target variable is used and the secret variable has a specific value. The advantage of the SEA is that it requires only a single-bit information from fault observation: If the faulty value has been used or not. Fault injection may be used to check if a specific computation is executed (C-safe errors [103]) or if a specific memory location is accessed (M-safe errors [15]). SEA attacks have been successfully demonstrated on symmetric-key [78, 104] and public-key [78, 103] algorithms.

Algorithm-specific Fault Analysis uses fault injection to exploit algorithm-specific properties. For instance, the public-key cryptography algorithms such as RSA and ECC rely on a hard-to-solve mathematical problem. An adversary may use fault injection to alter the mathematical foundations of the problem and convert the problem into an easy-to-solve one. Algorithm-specific analysis attacks have been mounted on several public-key systems including RSA and ECC [5, 78, 105, 106].

Biased Fault Analysis attacks [107, 108, 109, 110, 111] exploit biased fault behavior: Because of the correlation between the fault behavior of a target program and the applied physical fault intensity, the distribution of fault models is non-uniform. They allow an adversary to treat fault behavior as a side-channel signal, which relaxes the strict fault model requirements of the previous attacks.

2.6.3 Fault-Enabled Logical Attacks

In addition to their use in cryptanalysis, fault attacks can also be used to trigger logical attacks (e.g, control flow hijacking, privilege execution, subverting memory isolation, etc.) on smartcards and general-purpose processors. A typical logical attack (e.g, buffer overflow) tampers with the inputs of a program to exploit a security bug (e.g, memory safety bug) in the implementation of the program. In the absence of such an exploitable software bug, it is not possible for an adversary to mount a logical attack by just modifying inputs. In such a case, an adversary can inject faults to dynamically create required conditions to mount a logical attack. The following paragraphs briefly explain fault-enabled logical attack examples from the literature.

Barbu et al. [38] demonstrated two fault-enabled logical attacks on a Java card. In the demonstrated attacks, the adversary uses a laser-induced instruction-skip model to create type confusion. Then the adversary exploits the induced type confusion to load an unverified adversary-controlled code on the Java Card. Type confusion also enables an adversary to access other applications' memory space. Vetillard et al. [40] and Bouffard et al. [37] also demonstrates similar attacks on Java Card, in which they employed fault injection to bypass run-time security checks and execute malicious code on the platform.

The first fault-enabled fault attack on a general-purpose processor has been demonstrated by Govindavajhala et al. [85]. In the demonstrated attack, the adversary designs and runs a software program on a Java Virtual Machine (JVM) on a desktop computer. The malicious program is designed such that a bit error in the data space of the program allows the adversary take full control over JVM. To induce those exploitable faults, the authors overheat the memory chips.

Nashimoto et al. [95] proposed a fault-enabled buffer overflow (BOF) attack on a buffer overflow countermeasure, which limits input size. The authors demonstrated the proposed attack on an 8-bit AVR ATmega163 and a 32-bit ARM Cortex-M0+ microcontroller. Their fault models were single and multiple instruction-skip, which are induced by clock glitching.

Timmers et al. [18] demonstrated two ARM-specific, fault-enabled logical attacks which are based on setting the program counter (PC) of a microprocessor to an adversary-controlled value. The authors alter the execution of a memory-load instruction (i.e, instruction replacement) via voltage glitching to set PC to an adversary-controlled value. The authors provide two case studies to demonstrate the use of such an attack. In the first case, the authors bypass a secure-boot mechanism and run their own unverified program on the processor. In the second case, the authors subvert the hardware-enforced isolation mechanism of a Trusted Execution Environment (TEE) and run their code program with the highest privileges on the processor.

Vasselle et al. [112] demonstrated a fault-enabled logical attack on a Quad-core ARM Cortex-A9 processor, which bypasses secure boot mechanism and allows an adversary to get highest privileges on the processor. The authors achieved privilege escalation by resetting the privilege-level-specifying bit of the Secure Configuration Register of the processor via laser fault injection.

Timmers et al. [17] proposed three fault-attack enabled logical attacks on a Linux Kernel to gain kernel-level execution privileges. The authors demonstrated their attacks on an ARM Cortex-A9 processor through voltage glitching. In the demonstrated attacks, the authors request system calls from the user space, and then, inject faults during the execution of system calls for privilege escalation. The gained privileges may allow an adversary to run an arbitrary code on the device and access the

memory space of other applications.

The software-controlled fault injection methods such as triggering memory disturbance errors broaden the scope of fault attacks as they allow remote fault attacks. For example, in Rowhammer attacks [91], an adversary-controlled program (running in a user space) injects bit-flip faults into security-sensitive DRAM memory cells by repeatedly accessing nearby cells. In 2015, Seaborn [113] demonstrated two practical Rowhammer attacks. The first attack induces bit-flips to escape from Google Native Client (NaCl) sandbox. The second attack use bit-flips in DRAM for privilege escalation. Gruss et al. [23] successfully mounted a Rowhammer attack from web browsers on four off-the-shelf laptops. Similarly, van der Veen et al [25] achieved privilege escalation on Android-running mobile platforms. Razavi [92] demonstrated a Rowhammer attack in a cloud setting, in which a malicious virtual machine induce memory disturbance errors to gain unauthorized access to memory space of a co-hosted virtual machine. Kurmus et al. [93] and Cai et al. [90] demonstrated that software-controlled memory disturbance errors can be triggered on Multi-cell (MLC) NAND Flash memories to mount fault-enabled logical attacks.

Finally, Tang et al. [22] exploited security-oblivious dynamic voltage and frequency scaling (DVFS) interface to induce faults in a smartphone. They demonstrated two software-controlled fault attacks. The first attack allows a malicious user-space program to inject faults into the operation of an encryption program running in Trustzone environment and to reveal the value of secret key stored in Trustzone environment. In the second attack, an adversary bypasses an authentication mechanism running in Trustzone to load an unauthorized program into Trustzone environment. These two attacks show that fault injection may enable an adversary to subvert hardware-enforced isolation mechanisms such as ARM Trustzone.

2.6.4 Using Fault Injection to Assist Reverse Engineering

Another potential use of fault injection is to assist reverse engineering. San Pedro et al. [114] and Le Bouder et al. [115] employed fault injection to reverse engineer modified implementations of Sbox blocks of Data Encryption Standard (DES) algorithm. Similarly, Clavier et al. [116] use fault injection to reveal the specifications of an AES-like block cipher. Jacob et al. [117] induce faults into the execution of an obfuscated cipher and retrieve the secret key. Courbon et al. [118] demonstrated a method to reverse-engineer gate-level structure of a hardware implementation of Advanced Encryption Standard (AES) algorithm, in which laser fault injection and image processing are combined.

2.7 Comparison of Fault Attacks on Hardware and Software Secure Systems

Fault attacks on both hardware and on software systems require the knowledge of the target system's implementation. Having more knowledge of the target system increases the adversary's control on the induced fault effects, and thus, yields more efficient fault attacks. In practice, the fault manifestation and the propagation mechanisms in hardware and software systems are different. Therefore, mounting an efficient fault attack on a software system requires the knowledge of different abstraction layers than attacking a hardware system does.

In a hardware secure system, a security algorithm is mapped into a netlist of logic gates and registers. The execution of the mapped algorithm is embodied as a se-

quence of register-transfer operations scheduled over multiple clock cycles. Every bit-level operation of the algorithm thus maps into a particular clock cycle and register transfer. During a fault attack on the hardware system, an adversary will target a sensitive bit-level operation by selecting specific clock cycles and register-transfers, and by applying a suitable fault injection method. Next, the effects of the fault injection will be propagated into a faulty system output through register transfers with a dependency on the faulty register transfer. In order to mount a fault attack on such a hardware model, it is sufficient for the adversary to understand the hardware-level operation, and to apply generic, gate-level fault models.

In a software secure system, a security algorithm is implemented as a sequence of instructions executed by a microprocessor. Each instruction goes through the instruction-execution cycle, consisting of several steps. The number of steps in the instruction-execution cycle is architecture dependent, and it can vary considerably from one microprocessor to the next. However, we observe that the smallest execution step in software, the instruction, still corresponds to multiple register-transfers at the hardware level.

A fault injection in a microprocessor will affect the correctness of instruction execution. The effect of the fault will be propagated into a faulty system output by instructions that have data-dependencies or control-dependencies on the faulty instruction. Hence, knowledge of the microprocessor architecture is insufficient to mount a successful fault attack; the adversary also needs to understand the dependencies defined by the software. Furthermore, the precise effect of a fault on a microprocessor instruction depends on the type of the instruction.

We conclude that a fault model for a secure software system must capture both the hardware and software aspects of the fault behavior. For a given processor architec-

ture, it should show the potential fault effects for each instruction during each step of the instruction cycle. The fault effects may depend on the type of the instruction as well as on the step of the instruction cycle affected by the fault. Furthermore, in complex processor architectures, the execution of multiple instructions is overlapping. A single fault injection during a single clock cycle may affect sequential instructions. To analyze these complex effects, the adversary needs to understand the fault sensitivity of instructions with overlapping execution. In the microprocessor fault model proposed in this research, we provide support for overlapping instruction execution. We also introduce a micro-architecture aware fault attack methodology based on the proposed fault model.

2.8 Comparison of Fault-Tolerance and Fault-Attack-Resistance

Considering the requirements of modern embedded systems and issues of fault-tolerant-based countermeasures, a crucial step for effective countermeasures is differentiating *fault attack resistance* from *fault tolerance*:

The Goals: Fault tolerance aims at guaranteeing a certain level of correctness under the assumption of a general, often random fault model.

The aim of fault-attack resistance is supporting a given security policy against an adversary who applies focused, intelligent faults to break security mechanisms. In case of a malicious fault, one must take care of not disclosing any information about the secret data. Therefore, the fault must be detected soon enough.

The Faults: In fault tolerance research, considered faults are almost always non-

malicious, simple, sporadic, and transient that are caused by radioactivity or cosmic rays: single bit-flip errors in storage elements, referred as the SEU (Single Event Upset) model. This is a suitable model because flipping a single bit is indeed the most probable logical effect of a physical interaction between the hardware and low-energy ions [119]. Another cause of non-malicious run-time errors is DVFS (Dynamic Voltage and Frequency Scaling) techniques that are used to increase energy-efficiency of a microprocessor [59]. These errors are also generally single bit-flip errors because the microprocessor runs in the vicinity of its nominal operating conditions.

In fault attacks, the operating conditions of a microprocessor are controlled by an intelligent adversary. The injected faults are intentional rather than natural. The adversary controls the size and location of the injected faults. In addition, the adversary can inject the same faults repetitively and adapt the fault injection parameters to break the security. Therefore, fault-tolerant techniques are not sufficient to withstand such an adversary [11, 120].

The Countermeasures: The non-malicious, natural faults occur randomly. Therefore, the whole system must be protected with redundancy to achieve fault-tolerance.

In fault attacks, fault injections are not randomly distributed but are focused on the security-critical parts of the system. Therefore, the redundancy can be applied to only security-critical parts of the system. This significantly reduces the overhead.

Chapter 3

Fault Injection and Analysis Setup

In this section, we describe the measurement setup that enabled us to experimentally verify our techniques. Figure 3.1 shows the high-level diagram of our experimental setup. The setup consists of a PC, a Device Under Test (DUT), and a fault injection module. The PC manages the fault injection process by controlling and configuring both the fault injection module and DUT. The DUT (a LEON3 processor in this research) executes a cryptographic algorithm and sends a trigger signal to the fault injection module. Trigger signal synchronizes the fault injection module and the DUT. It enables us to have a cycle-accurate control over the glitch timing. The fault injection module consists of two blocks. The clock glitch controller keeps the glitch parameters and controls the clock glitch injector. A predefined number of clock cycles after the trigger signal, the clock glitch injector injects the glitch specified by the glitch controller.

A typical fault injection experiment consists of the following steps:

1. The controlling PC configures the fault injection module. It writes the number of wait cycles after the trigger event, the glitch width, and glitch offset into the clock glitch controller registers.

2. The controlling PC configures and initializes the DUT.

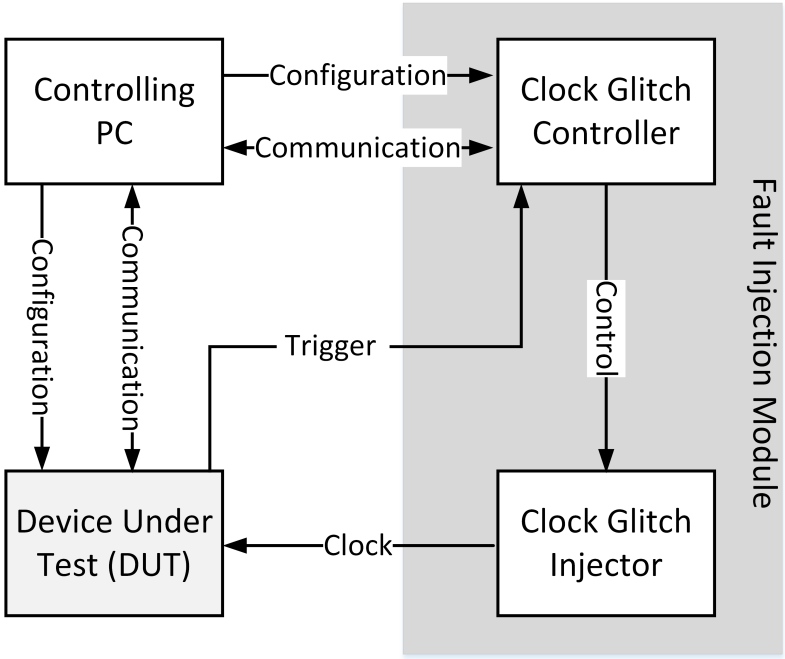


Figure 3.1: High-level block diagram of fault injection and analysis setup.

3. The controlling PC arms the clock glitch injector.

4. The controlling PC starts the execution of the DUT.

5. The DUT sets the trigger signal when it reaches the predefined point in its execution flow.

6. The clock glitch injector injects the glitch with the parameters specified in the Step 1.

7. The controlling PC read out the state and results of the DUT to analyze the effects of the injected glitch.

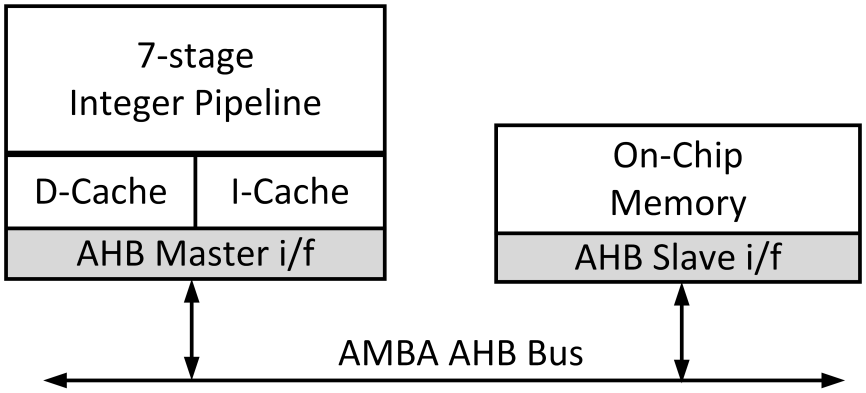


Figure 3.2: The LEON3 processor configuration used in this work.

3.1 The LEON3 Processor

The experiments in this paper are done on a 7-stage LEON3 RISC pipeline [121]. We will first review the main characteristics of LEON3.

LEON3 is a 32-bit SPARCv8-compliant RISC processor with (a modified) Harvard architecture, extensible through an AMBA 2.0 bus system. The core is distributed as a synthesizable VHDL model by Aeroflex Gaisler, together with a large library of peripherals and debug support modules. The core is highly configurable in capabilities and performance.

Figure 3.2 shows the block diagram of the LEON3 configuration used in our research. We use a configuration with a 7-stage integer pipeline and 64 KB of on-chip RAM memory. To evaluate cache effects, we also configured 4KB direct-mapped caches for Instructions and Data, each with a line size of 32-byte. We note that, although the LEON3 follows a Harvard architecture, both caches map into the same 64-KB on-chip memory.

For this paper, the seven stages of the LEON3 integer pipeline are of particular

interest. The purpose of each stage is as follows.

1. **Fetch (F):** An instruction is fetched from the memory or instruction cache, and copied into the instruction register (IR).

2. **Decode (D):** The instruction from the IR is decoded. For branch and CALL instructions, the target next-address is computed.

3. **Register Access (A):** The instruction operands are read from the register file or from internal forwarding paths.

4. **Executed (E):** ALU, logical, and shift operations are performed. For memory-load and store, the target data address is computed.

5. **Memory (M):** The memory-access part of an instruction is completed (data read or data write into memory).

6. **Exception (X):** This stage resolves traps and interrupts.

7. **Write-back (W):** The results of the instruction execution are written into the register file.

3.2 Setup Time Violation

In this research, we inject faults into the operation of a circuit by violating its setup time constraints. *Setup time violation* is a widely-used low-cost fault injection mechanism [122]. In the following paragraphs, we explain setup time constraints of a circuit and their use as the fault injection means.

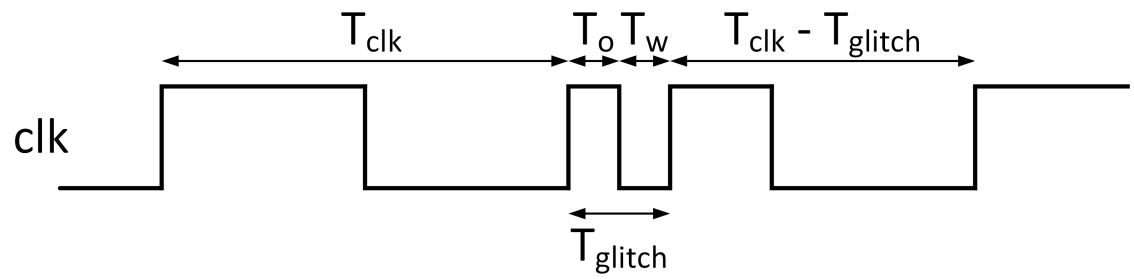


Figure 3.3: The effect of the glitch injection on the clock signal

In synchronous circuits the data is processed by combinational blocks, which are surrounded by input/output registers. The data is captured when the sampling edge of the clock signal arrives at the registers. Each combinational block requires a certain *propagation delay* (T_{pd}) to compute its output value. For the correct operation of the circuit, combinational block outputs must settle to their final values and remain stable at least some *setup time* (t_{su}) before the sampling clock edge. Therefore, the *clock period* (T_{clk}) must satisfy the following equation for all paths from input registers to output registers:

$$T_{clk} \geq T_{pd} + T_{su} \tag{3.1}$$

This equation specifies the *setup time constraints* of a circuit. The setup time constraint of the longest (i.e, critical) path determines the minimum clock period for the circuit. Applying a shorter clock period than this value will fail the setup time constraints.

We inject faults into the microprocessor by clock glitches. Figure 3.3 shows the effect

of a glitch on the clock signal. A clock glitch will temporarily shorten the clock cycle period from T_{clk} to T_{glitch}, thereby causing timing violation of the digital logic. When the *glitch period* (T_{glitch}) violates timing constraint of a path, the output value of this path is captured before its computation is completed. Therefore, the captured value is very likely to be faulty.

3.3 Implementation of Clock Glitch Injector

We implemented the fault injection module on the controller FPGA (Xilinx Spartan-6 XC6SLX9) of the SAKURA-G board [123]. To inject glitches into the nominal clock, we first generate two phase-shifted clocks from the nominal clock, and then, we combine these three clock phases (Figure 3.4 and 3.5) by applying logical operations [124], [16]. We use the Digital Clock Manager (DCM) blocks of the FPGA to generate the shifted clock phases. In this work, we use a 24-MHz nominal clock generated by an external pulse generator (Agilent 81110A). Using the dynamic phase-shifting feature of Xilinx DCM blocks and partial reconfiguration approach introduced by O'Flynn *et al.* [16], we can generate T_{glitch} values between 3ns and 20ns with 100ps step-size. We also have a control on the time between the trigger event and the glitch injection, which allows us to target a specific pipeline stage of a given instruction. We can dynamically set all of the glitch parameters via commands from the PC.

3.4 Implementation of Data Acquisition

Figure 3.6 shows the block diagram for the data acquisition part of our setup. Our DUT is a LEON3 processor, which is implemented on the main FPGA (Xilinx

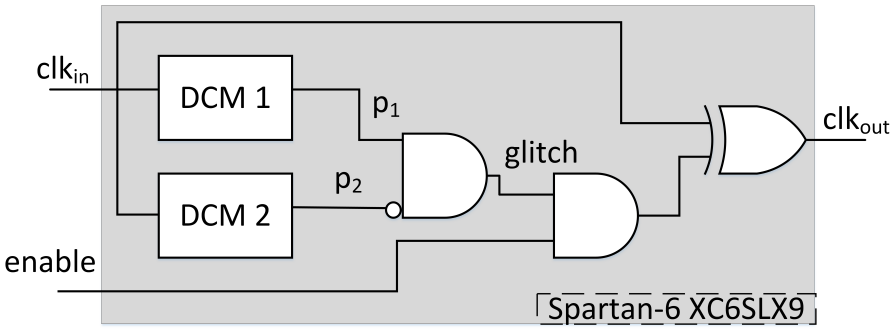


Figure 3.4: The block diagram of clock glitch generator circuit.

Spartan-6 XC6SLX75) of the SAKURA-G board. For data acquisition, we utilize three hardware blocks: Debug Support Unit (DSU) of LEON3, Instruction Trace Buffer (ITB) of LEON3, and an on-chip logic analyzer core (LOGAN) provided as a part of GRLIB IP library [125].

DSU is a non-intrusive on-chip debug core which controls the operation of the processor in the debug mode. In the debug mode, the processor pipeline is idle and the software-visible processor state can be accessed by DSU. DSU can read/write the architectural registers and memory locations, can load the program executable, can start the execution of a program, and can halt/continue the the operation of the processor. It can also set/use breakpoints and watchpoints.

ITB is a circular buffer that stores the executed instructions. It is located in LEON3 processor and read out via DSU core. It traces the instruction address, instruction result, load/store data and address, and timing information for the instruction. We use an 64-entry ITB for our experimental setup.

LOGAN implements an on-chip logic analyzer core and enables us to trace arbitrary signals inside LEON3 processor. It consists of a circular buffer to store the traced sampled signals, and a trigger block to detect a user-defined pattern on the sampled

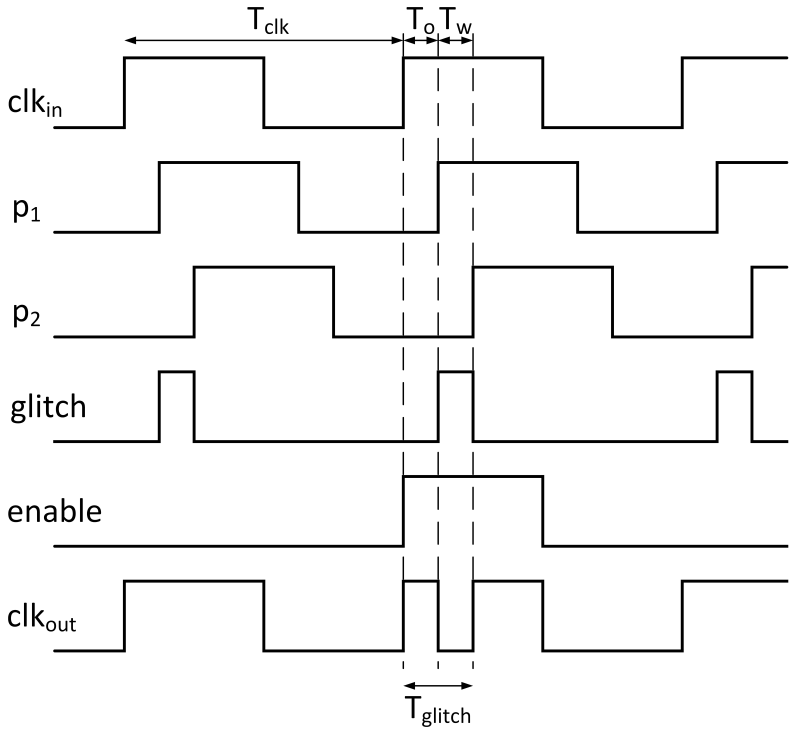


Figure 3.5: Clock glitch generation using phase-shifted signals.

signals. When it is armed, it continuously samples a set of signals until it detects a user-defined trigger condition. The trigger condition makes LOGAN stop tracing. Then the traced data can be read out. LOGAN core can store 4096 samples of 256 signals.

The controlling PC uses GRMON Debug Monitor [126] program to manage/configure the hardware data acquisition cores, to load the executable of the cryptographic software, to start the execution of the program, and read out the processor state. GRMON connects to the on-chip components via a JTAG Debug Link.

The DSU and ITB cores can only access the software-visible architectural registers and memory locations. Therefore, they can only show the fault effects on the

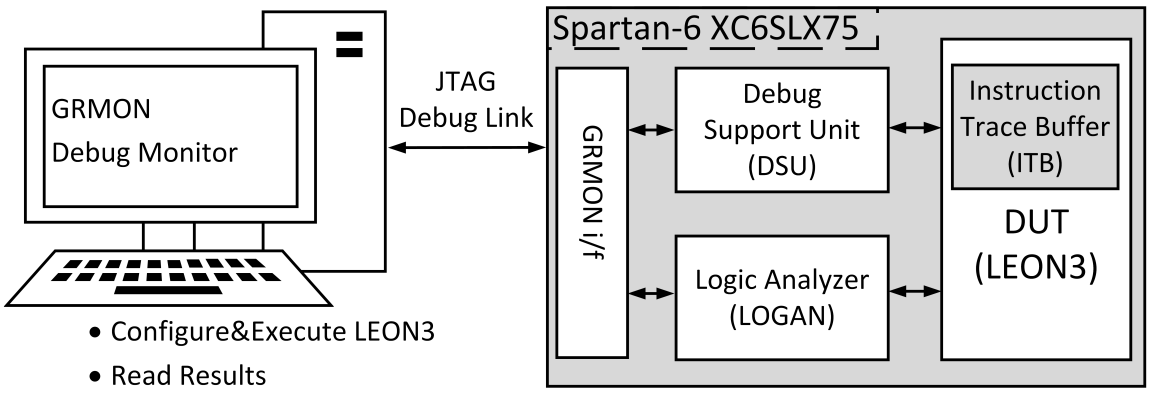


Figure 3.6: The block diagram of data acquisition part of the experimental setup.

software-visible architecture of the LEON3 processor. On the other hand, the LOGAN core can access any on-chip signal, and thus, it can provide information about the fault effects on the micro-architecture of the LEON3 processor. For instance, we can directly observe the fault effects on the pipeline registers through the LOGAN core.

In conclusion, we developed an experimental setup that enables high-precision fault injection and detailed analysis of the fault effects on the RISC pipeline.

Chapter 4

Instruction Fault Sensitivity Model

In this work, we introduce a instruction fault sensitivity model for a RISC pipeline. Figure 4.1 illustrates the relation of the proposed fault model to fault attacks. Without the model, traditional fault attacks such as DFA, C-safe/Errors or Flow Errors are still possible. With the fault sensitivity model, however, a new class of attacks based on fault bias becomes possible. These attacks include Fault Sensitivity Analysis (FSA) [77, 127], Non-Uniform Fault Analysis [110, 128], and Differential Fault Intensity Analysis (DFIA) [107].

We create the fault sensitivity model through timing simulation of a gate-level model of the RISC processor. When a cryptographic application executes on the RISC pipeline, the fault sensitivity model will show what instructions and what clock cycles are most suitable (or most sensitive) to fault injection, in order to mount a fault attack.

In the next sections, we present how to build an instruction sensitivity model for the LEON3 processor. We start with analyzing the fault behavior in a RISC pipeline.

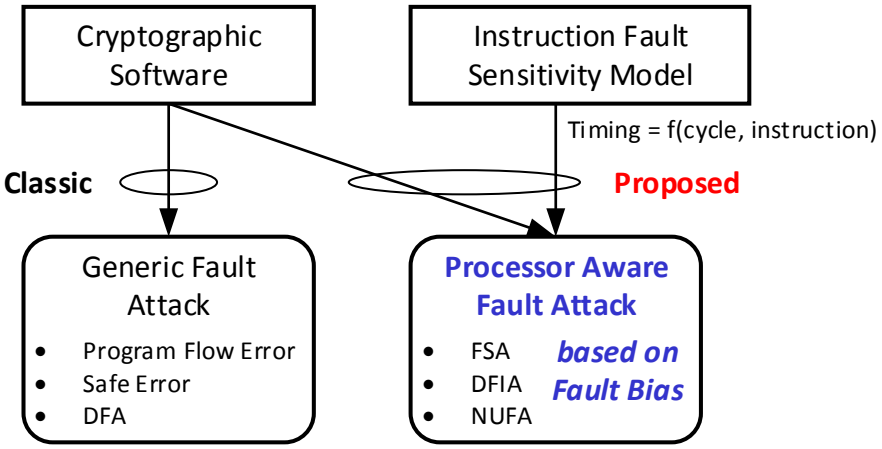


Figure 4.1: The proposed fault attack methodology uses a model of a RISC pipeline to enable more advanced fault attacks on software such as DFIA and FSA.

4.1 Fault Behavior in a RISC Pipeline

The RISC architecture is the dominant processor architecture in modern embedded systems, and therefore the analysis of this architecture is a meaningful starting point to develop a systematic fault model for software. In the following subsections, we provide a brief review of a typical RISC pipeline. Next, we define the generic fault injection mechanism that we will assume for the rest of the paper. We then describe the expected fault effects in a standard RISC-based stored-program architecture consisting of a pipeline and a memory. This leads to the classification of fault types utilized in our work.

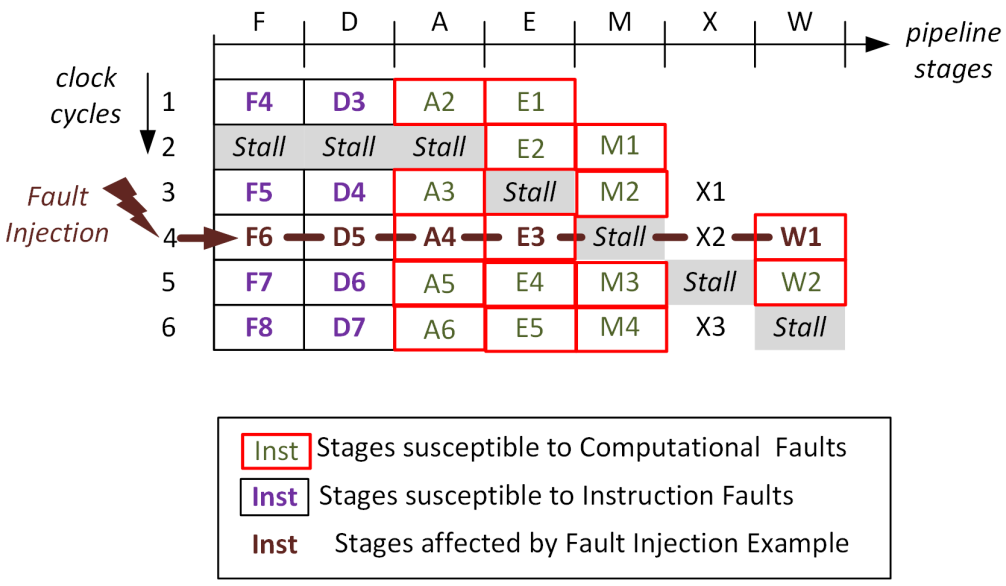


Figure 4.2: Fault propagation in a seven-stage RISC pipeline. A glitch can affect as many instructions as there are RISC pipeline stages. Careful control of the fault intensity limits the fault to the slowest pipeline stage. Furthermore, pipeline stalls will temporarily *blind* stalled pipeline stages from glitches.

4.1.1 Fault Injection in the RISC Pipeline

Figure 4.2 demonstrates the effect of a cycle-accurate fault injection on the execution of the 7-stage LEON3 pipeline. In any given clock cycle, multiple instructions execute simultaneously, each of them at a different step of the instruction cycle. Therefore, a fault injection can potentially affect as many instructions as the number of pipelines stages. However, the actual number of affected pipeline stages is determined by fault injection intensity, and by pipelining effects.

Pipelining effects such as data dependencies between the concurrently executed instructions, cache misses, and branch interlocks cause stalls in the pipeline. The stalled pipeline stages are not affected by the fault injection. For example, the fault

injection in cycle 4 in Figure 4.2 does not affect the Memory Stage.

Under a constant fault injection intensity, the fault effect will be different for different stages of the pipeline. This is because of variations in the detailed architecture of pipeline. Therefore, it is also possible that a fault injection at a specific fault intensity can affect some pipeline stages while leaving others untouched. A pipeline stage is affected by the fault injection only if the applied fault intensity is greater than the fault sensitivity for that pipeline stage. For example, consider again cycle 4 of the pipeline in Figure 4.2. By reducing the fault intensity, the adversary will eventually end up with a single faulty pipeline stage, namely the most sensitive pipeline stage. Assume for example that the execution stage would have the longest critical delay. Then, a fault injection with low intensity will affect only `E3`, the execution of instruction 3 in cycle 4. After such an isolated fault is injected, it will further propagate through the pipeline stages and affect instructions with dependencies on the faulty instruction 3.

Thus, we observe two important effects in a running RISC pipeline that provide an adversary with *additional* opportunities for controlled fault injection. The first effect is the pipeline hazard, which may depend on software as well as on the processor architecture. The second effect is the variation of fault sensitivity with RISC pipeline stage, which may provide the ability to target a single pipeline stage through a carefully chosen fault injection intensity. In our proposed attack methodology in Chapter 5, we utilize both of these effects to our advantage.

4.1.2 Instruction Faults and Computation Faults

As every instruction moves through the instruction-execution cycle, its fault behavior changes as a function of the step within the instruction-execution cycle. We partition the fault effects into two main categories.

1. *Instruction Faults (IF)* are faults that affect the control flow or instruction sequence of a program. Instruction Faults are created by fault injection in the fetch or decode pipeline stages (see Figure 4.2). For example, a fault could skip an instruction. An adversary can leverage this to break software fault countermeasures, by bypassing instructions that check integrity or branch when an error is detected. A fault could also change the meaning of an instruction by modifying the opcode. Balasch describes the effect of faults on the instruction fetch of an AVR microcontroller [29]. By gradually increasing the fault intensity, the opcodes of instructions are modified (as a result of timing violations), until they eventually turn into a `nop` instruction.

2. *Computational Faults (CF)* are faults that cause errors in the data used by a program. They are created by any error in the stages A, E, M, and W as indicated in Figure 4.2. An error in any of these stages can contribute a faulty value to the register file or memory. Faults in these stages will eventually propagate to the output, and they are suitable for traditional fault analysis mechanisms such as DFA or DFIA.

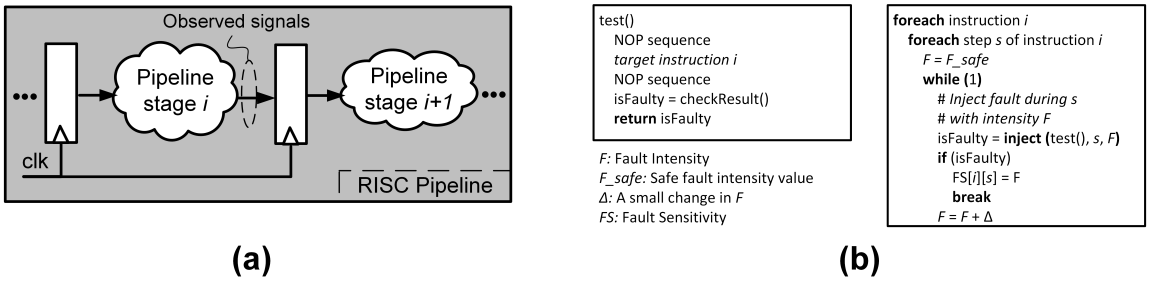


Figure 4.3: (a) The observed signals in the gate-level simulation. (b) A generic pseudocode for creating an instruction fault sensitvity model.

4.1.3 Fault Injection in the Memory

An alternate target for fault injection is the instruction-memory or data-memory used by a RISC processor. We can distinguish two types of memory faults: those that happen as a result of the direct execution of a RISC instruction, and those that happen independently of software execution. Our instruction fault model concentrates on the first case, and it can capture faults that occur as a result of instruction-fetch, operand-fetch, and result-writeback. On the other hand, in this research, we will not further develop fault attacks based on independent memory faults. We observe that error-check mechanisms are quite common in contemporary memory architectures - and this provides additional justification for our focus on processor faults.

4.2 Timing Characterization of RISC Pipeline

In this section, we provide an example instruction fault sensitivity model for a subset of SPARC instructions. We characterized the fault sensitivity of instructions on a LEON3 processor for setup-time violation based faults. We implemented LEON3 processor on a Spartan 6 FPGA(45nm technology), and we characterized the fault

sensitivity by applying gate-level simulation on the post place-and-route model of the implementation.

For a given target instruction, we characterize the timing of each pipeline stage by extracting its critical path delay via gate-level simulation. First, we simulate the microprocessor while it is running a test program. The program includes the target instruction surrounded by *nop* instructions. Then, we analyze the signal activity within the fan-in cone of pipeline registers (see Fig 4.3a) to determine critical path values. Repeating this analysis for each target instruction, we obtain the microprocessor fault sensitivity model. The model consists of the critical path information of each pipeline stage for each target instruction. Figure 4.3b illustrates the procedure to generate instruction fault sensitivity model.

We used combinational path delays as the fault sensitivity metric because it has been shown that there is a significant correlation between the path delays of an implementation and its sensitivity to setup-time violation attacks [81, 129]. Gate-level simulation has been employed earlier to create a better fault sensitivity models of AES ASIC implementations [130, 131]. However, we used the gate-level simulation for characterization of individual processor instructions.

Fault sensitivity of an instruction is not constant. Rather, it varies as a function of the pipeline stage. We verified this by extracting the combinational delay of individual pipeline stages for each instruction. For each target instruction, we wrote a test program and applied a fault sensitivity characterization algorithm as shown in Figure 5.1. The test program includes the target instruction surrounded by two sequences of `nop` instructions. For a test program, we characterized the sensitivity of a pipeline stage by injecting faults with varying fault intensities during the execution of this specific stage: Starting from a safe fault intensity value, we gradually increased

Table 4.1: LEON3 Instruction Fault Sensitivity Model (Critical Delay in ns)

Instruction	**F**	**D**	**A**	**E**	**M**	**X**	**W**	Meaning
`xor reg, reg, reg`	5.54	3.72	6.52	5.12	0	3.26	4.92	Bitwise xor
`ld mem, reg`	5.72	3.36	5.4	5.2	7.58	2.82	5.6	Load from memory
`ldi mem, reg`	3.53	3.26	5.62	5.2	7.58	3.17	4.45	Load Indexed
`st reg, mem`	5.91	3.5	5.78	5.37	5.35	3.77	0	Store Word
`sll reg, imm, reg`	5.53	3.26	5.2	5.09	0	2.16	5.6	Shift Left Logical
`srl reg, imm, reg`	5.91	2.83	4.7	5.67	0	3.25	5.61	Shift Right Logical
`cmp reg, reg`	6.86	5.53	4.98	5.23	0	4.78	5.6	Compare
`bne add`	6.40	5.00	5.85	3.92	2.020	0	5.6	Branch on Not Equal
Fault Type	**Instr Fault**		**Computation Fault**					

the fault intensity until we observe a faulty output. Repeating this analysis for each target instruction, we obtained the instruction fault sensitivity model.

In our simulations, we simulated each instruction one time with arbitrarily selected operands to accelerate the characterization process. We assumed that the impact of the operand values on the variation of instruction timing is smaller than the impact of the instruction type and the pipeline stage. We made this assumption based on the previous work [14, 29]. For example, Korak et al. analyzed two different microcontrollers (an ATMega256 and an ARM Cortex-M0) against voltage and clock glitching. They observed different fault intensity intervals for different instructions and pipeline stages. They showed that the decode stage of their target was always fault-free while they were able to create faults in fetch and execute stages. Their results also demonstrated that execution stages of different instructions have different timing values. One can also run each instruction with multiple data sets and apply statistical processing of the results to create the instruction fault sensitivity model. Such an approach would capture the variation effects in a more accurate way while increasing the characterization time.

Table 4.1 illustrates the obtained instruction fault sensitivity model. The figures in

the table cells indicate the critical delay (in ns) of the instruction as it flows through each pipeline stage. One can clearly distinguish a variation in timing horizontally, across a single instruction. This confirms that the critical delay varies as a function of the pipeline stage. In addition, one can see a timing variation vertically, across a column for a particular pipeline stage. This means that different instructions have a different timing requirements. Now, imagine a running software program; the pipeline is filled with instructions as illustrated in Figure 4.2. Using the instruction fault sensitivity model, it is clear that we can predict which pipeline stages will be faulty when we inject a fault with a given fault intensity. This will be the basis of the fault attack methodology, explained in the next subsection.

The obtained model captures several fault injection methods because setup-time violation faults can be achieved by various techniques including clock glitching, voltage glitching, voltage underfeeding, overheating, and electromagnetic pulses [81, 132]. For other fault injection methods such as laser pulses, the same approach can be used to obtain a similar instruction sensitivity model. In addition, the same algorithm can also be used to build instruction fault sensitivity model with physical fault injection. If an adversary has a COTS implementation of the target and a fault injection setup, the adversary can use the same test program and algorithm to create an instruction fault sensitivity model.

Chapter 5

Micro-architecture Aware Fault Injection Attack (MAFIA)

In the previous section, we create an instruction fault sensitivity model to capture fault effects on the software. This section proposes an *instruction fault sensitivity model* and a fault attack strategy *Microarchitecture Aware Fault Injection Attack (MAFIA)* based on the instruction fault sensitivity model. As it is shown in Figure 5.1, the proposed methodology consists of two phases: (i) Characterization Phase and (ii) Attack Phase.

The purpose of the *characterization phase* is to create an instruction fault sensitivity model for the target processor. The instruction fault sensitivity model shows the fault sensitivity for the steps of each instruction. A processor potentially carries out each step of an instruction is on a different microarchitectural block. Therefore, one can also think of the instruction fault sensitivity model as the characterization of main microarchitectural blocks for each instruction. The key point is that the fault sensitivity is instruction-dependent because each instruction uses a specific subset of the available microarchitectural blocks. The instruction fault sensitivity model can be generated once for a specific fault injection method and a target processor, and it can be used multiple times for different attacks on the target processor.

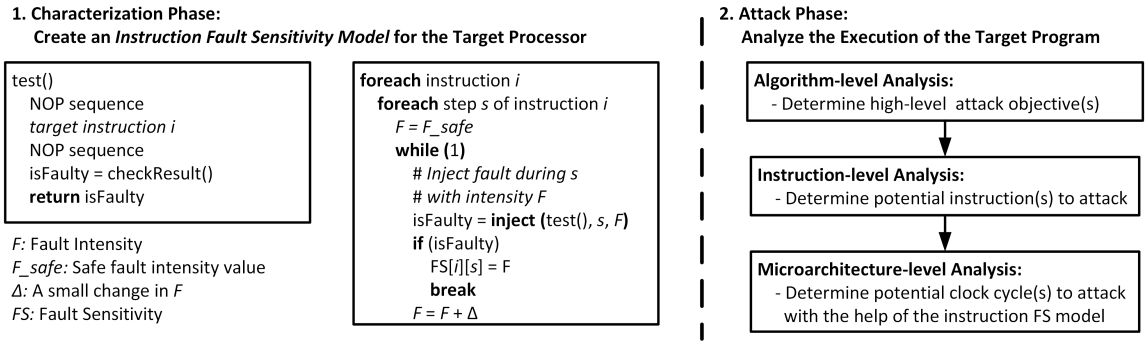


Figure 5.1: Overview of Microarchitecture Aware Fault Injection Attack (MAFIA): 1- An instruction fault sensitivity model is built for the target processor. 2- The created model is combined with the analysis of the software program to determine the best fault injection parameters.

The purpose of the *attack phase* is to design and implement a fault attack on a target program running on the characterized processor. The adversary analyzes the target program at different abstraction levels, and then, combines this analysis with the instruction fault sensitivity model to determine the best clock cycle(s) and processor part(s) to attack. The attack phase consists of three steps. An adversary first analyzes the target program at the *algorithm-level* to determine the application-specific attack objectives. Then the adversary applies an *instruction-level* analysis to determine potential instructions to attack in the program flow. In the last step, the adversary analyzes the execution of the software program on the target micro-architecture to identify potential clock cycles to attack. Finally, the adversary uses instruction fault sensitivity model to determine fault injection parameters for each potential clock cycles to attack. As the result, the adversary has a set of (*clock cycles*, *fault parameters*) pairs to attack. Using this information, the adversary can carry out the actual fault injection experiments.

The proposed method requires ISA-level, microarchitecture-level, and hardware-level

knowledge of the target system. The ISA-level knowledge is required to understand specifications and semantics of the instructions. The adversary can use open-source architecture manuals to acquire this knowledge. The microarchitectural-level knowledge enables the adversary to understand how a given instruction is executed on the target platform. The adversary may have an instruction-accurate model(e.g, a software-level emulator such as QEMU [133]), a cycle-accurate model (e.g, a micro-architecture simulator such as gem5 [74]), a register-transfer level model, or a commercial-of-the-shelf (COTS) implementation of the processor. The hardware-level knowledge is necessary to determine the sensitivity of the processor hardware to the physical fault injection. The adversary can gather hardware-level information from the gate-level netlist, transistor-level netlist, layout, register-transfer level definition, or a COTS implementation of the processor hardware. As a result, an adversary's knowledge level may vary from zero-knowledge to full-knowledge in practice. The key point is that a fault attack would be more efficient if an adversary has more knowledge of the implementation. The proposed method provides a methodology to systematically combine the knowledge of different abstraction layers.

Next, we explain the details of the proposed methodology by means of an example. This illustrates how the proposed instruction fault model is used in practice. We follow the assumption of a conventional, pipelined RISC microprocessor. Instructions flow through seven stages of a pipeline and can be stalled because of hazards. In addition, the effects of hazards can be minimized through conventional techniques such as forwarding, or careful instruction scheduling.

5.1 How MAFIA Works

This section demonstrates how the instruction fault-sensitivity model can be used to design an attack on a software secure system. We will describe a generic methodology, which will be applied on specific cases in later sections. The methodology has three phases, and we will briefly describe each of them through an example.

5.1.1 Algorithm-level Analysis

Initially, the adversary investigates the software implementation at the highest level of abstraction available. This could be C source code, or a binary of the software secure system. During algorithm analysis, the adversary defines the application-dependent attack objectives. For example, the differential fault analysis attack of Piret [134] requires a fault differential on the ciphertext, obtained after injecting a fault into the last-round state of block cipher. In the following example, we assume a last-round computation of the form below. The fault attack target, in this case, is the `state`, and the objective is to capture a faulty `ciphertext`.

```
ciphertext = key xor sbox[state]
```

5.1.2 Instruction-level Analysis

Next, the adversary studies the software implementation at instruction-level, in order to identify the candidate instructions for the fault attack. Consider the following example, which is a simplified implementation of the last-round of a block cipher, implementing using LEON3 instructions.

```
LD      [%fp - 12], %g1      ;LD1 (state)
LD      [%fp - 16], %g2      ;LD2 (key)
LD      [0x100 + %g1],%g3    ;LD3 (lookup[state])
XOR     %g3, %g2, %g4        ;XOR1 (key, state)
STD     %g4, [%fp - 20]      ;STD1 (output)
```

This program snippet loads a state variable and a key from data memory, substitutes the state variable using a lookup table, and XORs the result with the secret key. The output is then transferred to a memory location. Based on the DFA technique of Piret, the candidate instruction for fault injection is the first load instruction, which transfers the last-round state from data memory into processor register `%g1`. Additional study of the control-flow and data-flow of the instructions may be required in order to identify dependent instructions which are also suitable for the fault attack. In this case, only the first load `LD1` is a suitable fault-injection candidate. The second load, `LD2` does not contribute to the desired fault analysis, and faults in the third load `LD3` would cause faults at the output of the lookup table rather than at the input.

5.1.3 Microarchitecture-level Analysis

The third step is to examine the software behavior closely, in order to select the exact clock cycle that would lead to the desired error. One would need to do this analysis at the cycle-accurate level, in order to account for pipeline effects. Figure 5.2a illustrates the result of this analysis. The instruction sequence requires 12 clock cycles to complete and experiences one data hazard, to accommodate the table lookup. According to the pipeline analysis, `LD1` occupies seven clock cycles and a

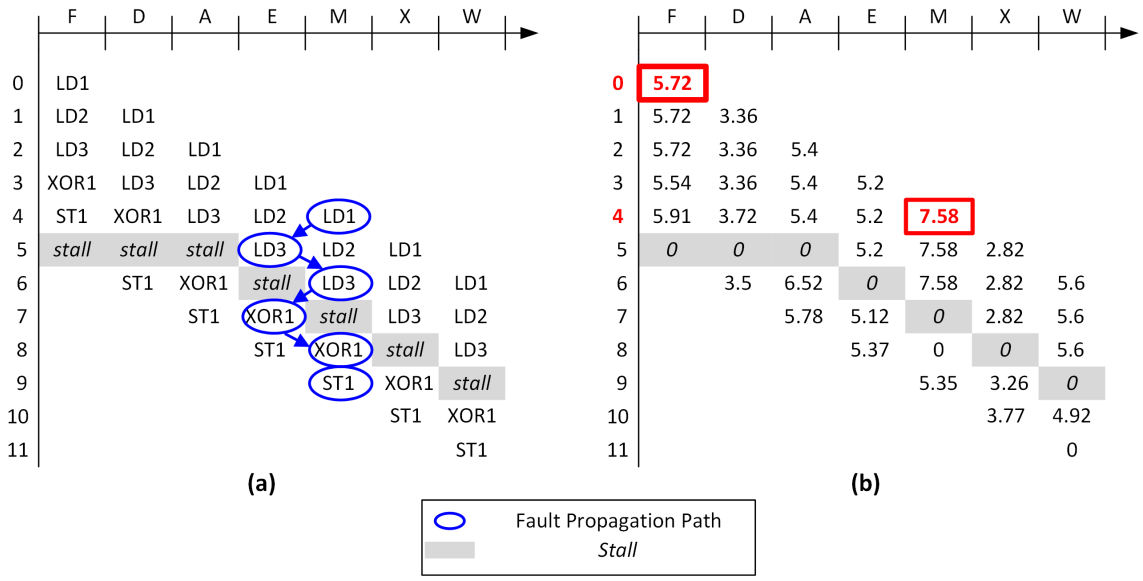


Figure 5.2: (a) Pipeline analysis and fault propagation path for the lookup table target. (b) Pipeline sensitivity analysis for the lookup table target.

fault in any of these seven cycles can potentially create a fault of the desired kind.

However, not all seven cycles can lead to useful `LD1` errors. Some of the fault injections may also affect other instructions. We can use the instruction fault-sensitivity model to identify the cycle most suited for fault injection. Figure 5.2b shows an overlay of the instruction fault sensitivity data (Table 4.1) on the instructions in the pipeline. Looking across rows, one can now identify the most sensitive pipeline stage in every clock cycle. For the `LD1` instruction, the instruction-fetch and memory-access step are most sensitive, which implies that clock cycle 0 and clock cycle 4 are suitable candidates for fault injection. The fault intensity would have to be chosen such that a fault is induced in `LD1` but not in other instructions. Assuming that we would select cycle 4 for fault injection, Figure 5.2a shows the fault propagation to the memory stage of the final instruction `STD`. As we will demonstrate in the next

sections, the analysis of the fault propagation path may sometimes lead to additional candidates for fault injection.

5.2 Case Studies: Fault Attacks on Secure Embedded Software

In this Section, we will apply the principles of the proposed fault attack methodology on two case studies - one involving a DFIA analysis on AES, and the other involving an attack on software-implemented fault countermeasures.

5.2.1 Case Study I: DFIA on TBOX AES

This section builds a Differential Fault Intensity Attack (DFIA) on the software implementation of the AES algorithm on LEON3. We implemented the AES algorithm as a TBOX design. TBOX design is a word-oriented implementation suited for 32-bit microprocessors. The details of the DFIA attack on the AES algorithm can be found in [107].

Algorithm-level Analysis

The objective of the DFIA attack on AES algorithm is to mount a biased fault injection attack on the output of AES round-9. For the attack on the TBOX design, we consider the expression that generates (one quarter of) the round-9 output state. It includes four TBOX-table lookups, which are all added together with a roundkey to produce the round-9 output `t0`.

```
//Target for biased fault injection: Output of Round 9
t0 = Te0[ s0 >> 24    ] ^
Te1[(s1 >> 16) & 0xff] ^
Te2[(s2 >>  8) & 0xff] ^
Te3[s3 & 0xff        ] ^
rk[36];
```

Instruction-level Analysis

Now, we must define the set of instructions that would lead to the required fault model. The code below shows some instructions in the `AddRoundKey` function of the TBOX AES in round-9. The instructions in range of $[48, 5c]$ are the instructions that are doing shift and XOR operations on the state word (`%g1`). Instruction *LDI7* loads the key word, and instruction *XOR8* applies a bitwise XOR operation on the key and the state words.

```
48: LD   [%l3 + %o3], %o2     ;LD1
4C: SLL %o4, 2, %o4           ;SLL2
50: LD   [%o7 + %o4], %o5     ;LD3
54: XOR %g1, %o2, %g1         ;XOR4
58: LD   [%fp + 0x4c], %l2    ;LD5
  5C: XOR %g1, %o5, %g1         ;XOR6
60: LD   [%l2 + 0x98], %o4    ;LD7
64: XOR %g1, %o4, %l6         ;XOR8
68: SRL %i5, 0xe, %o5         ;SRL9
6C: SRL %i4, 0x18, %g1        ;SRL10
```

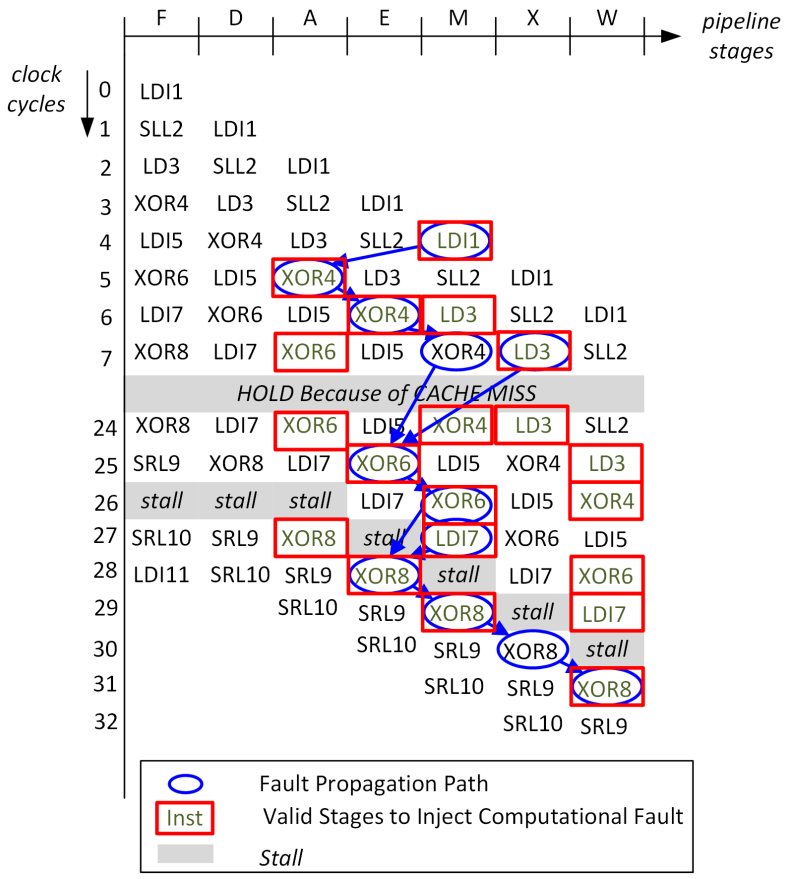


Figure 5.3: Pipeline Behavior for AddRoundKey of Tbox AES

Microarchitecture-level Analysis

To apply DFIA to a RISC pipeline, we need to induce biased data errors into selected instructions. Biased data errors mean that the effects of the injected fault must be revealed on the data computation rather than the control flow of the program or the instruction opcodes. Therefore, we will consider a (*pipeline stage, instruction*) pair as a valid fault injection target if attacking that pair yields a biased fault.

Figure 5.3 shows the behavior of the previously analyzed instructions in the pipeline.

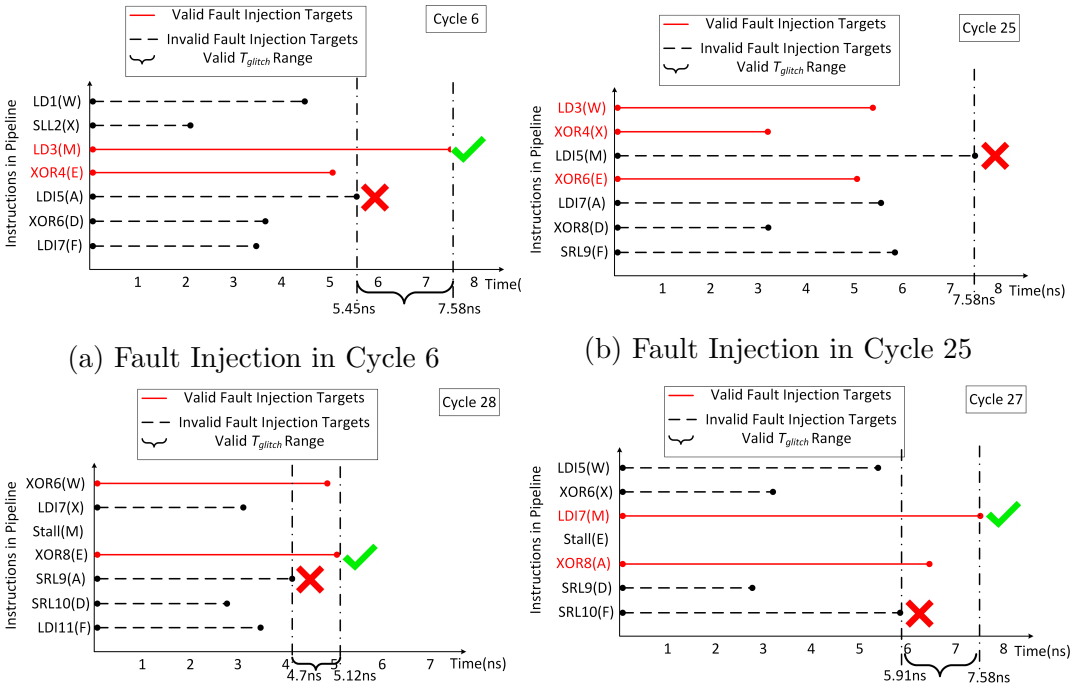
There are some data dependencies between the instructions for TBOX as well which are shown by blue circles in the graph. All the data dependencies can be solved by forwarding technique, except the dependency from LDI7 to XOR8. Therefore, the pipeline will have a stall in cycle 26. Due to the cache miss processing for LDI5, the pipeline will be held still until the data is ready. The HOLD cycles extend the time of the instruction causing the miss by the corresponding number of cycles. The potential points of observing biased fault is shown by red circled instructions in the pipeline in the Figure 5.3.

In order to perform an efficient attack and obtain useful faulty values, the adversary must only affect the circled pipeline stages only and avoid affecting other pipeline stages in a cycle. In order to find the fault injection location and the valid fault intensity range, we need to look into the timing characterization of each of the potential targets.

There are 32 opportunities to inject fault into AddRoundKey of TBOX since we can inject fault in any cycles of this function. In Figures 5.4a- 5.4d, we show four of these targets. Based on analyzing the instructions that might be affected by each of these targets, we will choose the specific target and fault intensity to inject the fault.

1. *Fault Injection in Cycle 6* (Figure 5.4a): The only blue operations in cycle 6 are the XOR4(E) and LD3(M). Figure 5.4a shows the length of critical path for each operation in cycle 6. The largest critical path is for LD3(M), which is equal to 7.58ns. By gradually increasing the fault intensity, we will affect the LDI5(A) which is 5.45ns. Since LDI5(A) is invalid fault injection target, Figure 5.3, the maximum fault intensity can be 5.45ns.

2. *Fault Injection in Cycle 25* (Figure 5.4b): By injecting fault in this cycle, we



(a) Fault Injection in Cycle 6

(b) Fault Injection in Cycle 25

(c) Fault Injection in Cycle 27

(d) Fault Injection in Cycle 28

Figure 5.4: Fault Sensitivity Analysis for Different Target Cycles of the TBOX AES

are allowed to affect XOR6(E), XOR4(X) and LD3(W). The largest critical path belongs to LDI5(M). Since LDI5(M) is not blue, we cannot inject fault into this cycle.

3. *Fault Injection in Cycle 27* (Figure 5.4c): In this figure, we are allowed to affect XOR8(A) and LDI7(M). We can only affect LDI7(M) with the fault intensity of 7.58ns. The maximum fault intensity will be 5.91ns since we do not wish to affect the SRL10(F).

4. *Fault Injection in Cycle 28* (Figure 5.4d): In this figure, we are allowed to effect XOR8(E), XOR6(W). We can affect the XOR8(E) and the maximum intensity that we can apply is the delay of SRL9(A) (4.7ns), since SRL9(A) is

invalid target of fault injection in Figure 5.3.

Using this analysis, we launched the DFIA attack on the TBOX implementation of the AES algorithm. The number of required fault injections and the results of this attack will be discussed in Section 5.3.

5.2.2 Case Study II: Analysis of Instruction-level Countermeasures on LEON3 Pipeline

The redundancy-based, instruction-level countermeasures against fault attacks have been trusted for years. Traditionally, it is assumed that breaking these countermeasures requires multiple identical faults, which can be achieved by expensive fault injection setups with back-to-back injection capabilities [46, 66]. In this section, to break common instruction-level countermeasures, we describe different attack scenarios that can be potentially achieved by single glitches without requiring back-to-back injections. We will use microarchitecture-level analysis to identify the attack scenarios. In Section 5.4, we experimentally verify the defined scenarios with single clock glitch injections.

Instruction Duplication Countermeasure

In the Instruction Duplication countermeasure [66], sensitive instructions are duplicated selectively and their computation results are stored in different destinations. Then, these results are compared to detect faults.

Figure 5.5 illustrates the Instruction Duplication countermeasure on a Memory Load (`LD1`) instruction and its behavior on the pipeline. This code protects the `LD1` in-

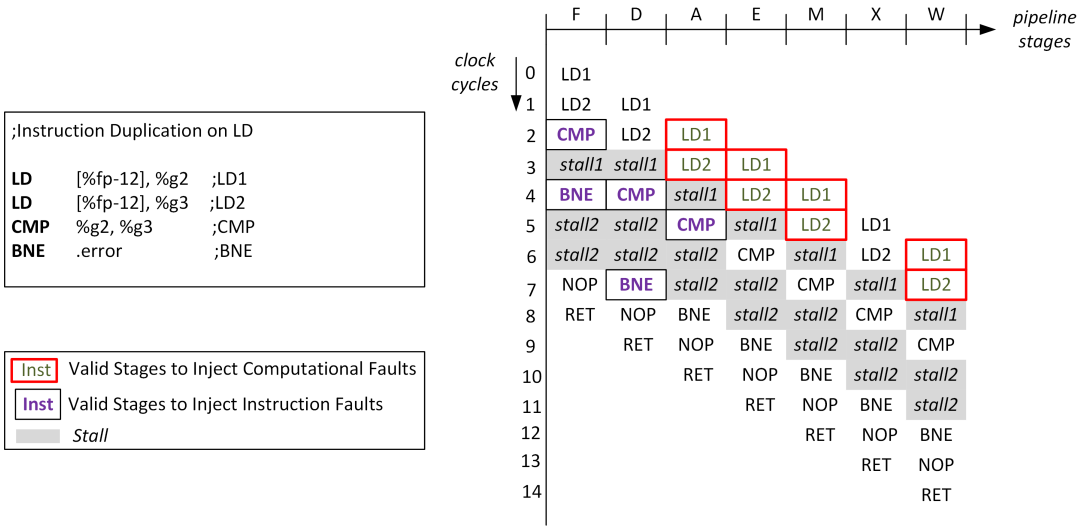


Figure 5.5: Pipeline Behavior for Instruction Duplication Countermeasure for LD Instruction

struction by storing the load value in both `%g2` and `%g3` registers. The values of these two registers are compared by `CMP` instruction and an error policy is called if a mismatch is detected.

The gray instructions show stalls in the pipeline. *Stall 1* in Figure 5.5 is due to the data dependency between the `LD2` and `CMP` instructions. As shown, the results of `LD2` will be ready in the *Stage E* at *Cycle 4*. The `CMP` instruction in *Stage A* waits for the result of `LD2` until *Cycle 5* and then continues its execution. *Stall 2* on the `BNE` instruction is due to the branch interlock. The branch interlock happens in the case of a conditional branch. When a conditional branch is performed in 1-2 cycles after an instruction which modifies the condition codes, 2 cycles of delay is added to allow the condition to be computed.

The target of fault injection to thwart this countermeasure can have two forms. First, injecting two identical faults in each of the `LD` instructions to bypass the equality

check. Second, a single fault injection into the `LD1` instruction and skipping the `CMP` or `BNE`. In this section, we use the pipeline analysis of the Instruction Duplication code to find vulnerable points of fault injection to create different scenarios explained above. The first step is to define the valid targets of fault injection in each cycle. Based on the previous analysis, the valid stages to inject faults are defined as follows. The red circled instructions show the valid instructions to target computation faults. These instructions can be targeted to generate different faulty values in registers. Affecting the black squared pairs of (*instruction, pipeline stages*) will cause instruction faults. For example, targeting the `LD` instructions in *Stage M* or *W* stage will generate different faulty values and targeting the `CMP` or `BNE` instruction in *Stage F* and *Stage D* will cause instruction faults. Then, an adversary can define different scenarios for fault injection in each cycle.

Table 5.1 summarizes the two potential scenarios. In this table, columns 2 and 3 show the potential targets of fault injection in the pipeline for achieving each scenario. The last column shows the type of fault that is injected into each instruction of the pipeline. The two scenarios are explained in detail as follows.

1. **Scenario A.1. Double Computation Fault**: The purpose of this fault injection is to inject exactly the same faults into the original and redundant copies of the `LD1` instruction. These fault injections must not affect the `CMP` or `BNE` instructions. This scenario can be achieved by injecting a fault into *Cycle 3* of the pipeline. Injecting a fault into this cycle does not have any effect on the `CMP` or `BNE` instructions, due to *Stall 1* in the pipeline.

2. **Scenario A.2. Single Computation Fault-Single Instruction Fault**: Another way to bypass this countermeasure is to create faulty values in register

Table 5.1: Fault Attack Scenarios to Thwart Instruction Duplication Countermeasure

Scenarios	# of Glitch Injections	Targeted Cycles	Instruction, Fault Type			
			LD1	LD2	CMP	BNE
A.1.	1	3	CF	CF	-	-
A.2.	1	2	CF	-	IF	-
		4	CF	-	IF	-
			CF	-	-	IF

`%g2` by a computational fault in `LD1` and skip the `CMP` or `BNE` instructions. To achieve this type of fault, we can trigger the fault injection in different cycles. The single glitch injection must accurately target the cycle that is performing both computational operations on `LD1` and instructional operations on `CMP` or `BNE`. As shown in the pipeline, these cycles can be *Cycle 2* that affects `CMP(F)` or *Cycle 4* that can affect `CMP(D)` or `BNE(F)`.

Instruction Parity Countermeasure

This software countermeasure is proposed by Barenghi et. al in [66]. In this technique, we first save the precomputed value for the parity bit in a register. Then, the parity is computed on the fly for the protected register's value. The computed parity value is compared to the precomputed value and an alarm is raised if a mismatch happens.

Figure 5.6 shows an example of the parity countermeasure. In this example, the protected instruction is a Memory Load instruction that loads a value from `[%fp-12]` into register `%g3`. The precomputed parity value is stored in `%g2`. The computed parity value is obtained using some Shift-Right (SRL) and XOR instructions and

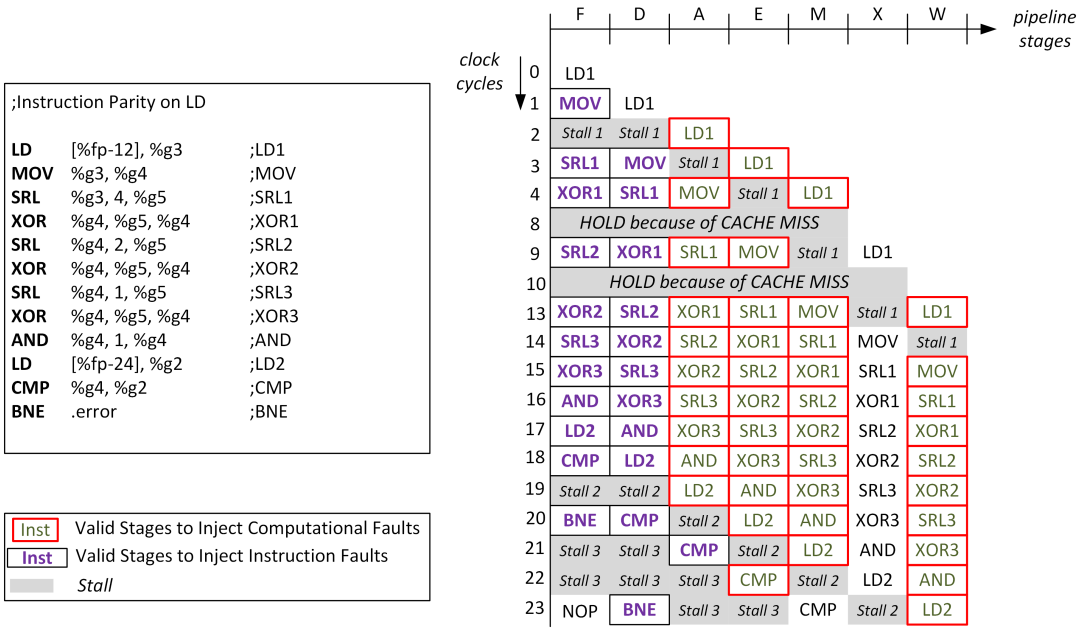


Figure 5.6: Pipeline Behavior for Instruction Parity Countermeasure for LD Instruction

stored in `%g4`. The value of the precomputed parity bit will be compared to the value of `%g4` and the countermeasure will raise an alarm in case of mismatch.

As shown in Figure 5.6, there are several opportunities to inject useful faults into this countermeasure. The parity countermeasure is vulnerable to many types of fault injection scenarios as it contains many instructions for computing the parity. Some of these opportunities are explained below, that is summarized in Table 5.2.

1. **Scenario C.1. Single Computation Fault**: The purpose of this fault injection is to inject computational faults into the original `LD` instruction, so that the effects of fault can change an even number of bits in register `%g3`. Therefore, the computed value for parity bit will still be the same as the correct execution. These fault injections must not affect the `CMP` or `BNE` instructions.

Table 5.2: Fault Attack Scenarios to Thwart Instruction Parity Countermeasure

Scenarios	# of Glitch Injections	Targeted Cycles	Instruction, Fault Type				
			LD1	SRL1-AND	LD2	CMP	BNE
C.1.	1	2,3,4,13	CF	-	-	-	-
C.2.	1	3,4,13	CF	IF	-	-	-
C.3.	1	2,20	CF	-	-	-	IF
			CF	-	-	IF	-

This scenario can be achieved by injecting a fault into the *Cycle 2, 3, 4 or 13*.

2. **Scenario C.2. Single Computation Fault-Multiple Instruction Fault**: The purpose of this fault injection is to inject computation fault into the `LD` instruction and inject instruction faults in the computation of the parity bit. This scenario can be obtained by a single glitch injection. The single glitch injection must accurately target the cycle that is performing both computational operations on `LD3` and instructional operations on `SRL1(F-D-E)`, `MOV(D-M)`, `XOR1(F-A)`, `XOR2(F)`, `SRL2(D)`, `XOR1(F)`.

3. **Scenario C.3. Single Computation Fault-Single Instruction Fault**: Another way to bypass this countermeasure is to create faulty values in register `%g3` by a computational fault in `LD1` and skip the `CMP` or `BNE` instructions. To achieve this type of fault, we should trigger the computation fault in *Cycle 2* and the instruction fault in *Cycle 20* to target the `BNE(F)`.

Instruction Skip Countermeasure

This countermeasure is proposed by [46]. This redundancy technique is used to avoid the instruction skip faults. Instruction skip fault is defined as a fault that can change an instruction to an effective `NOP` instruction. Therefore, the fault model is

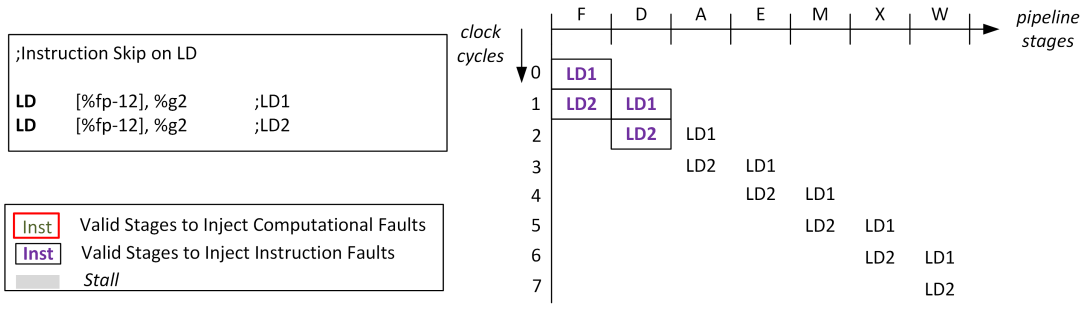


Figure 5.7: Pipeline Behavior for Instruction Skip Countermeasure for LD Instruction

Table 5.3: Fault Attack Scenarios to Thwart Instruction Skip Countermeasure

Scenarios	**# of Glitch Injections**	**Targeted Cycles**	**Instruction, Fault Type**	
			LD1	**LD2**
D.1.	1	1	IF	IF

not defined as computational fault model, but it can only change the opcode of the protected instruction. Following shows the assembly code for the instruction skip countermeasure on `LD` instruction.

The pipeline behavior for the instruction skip countermeasure is shown in Figure 5.7. As shown, since the adversary does not target for computational faults, he can only target the first three cycles. Therefore, there is only one scenario that can achieve such a fault.

1. **Scenario D.1. Double Instruction Fault**: There are two ways to achieve instruction faults in both `LD` instructions.

   - *Single Fault Injection*: By injecting a fault into *Cycle 1*, the adversary can change the opcode of the instruction in the `LD2(F)` and `LD1(D)` (as shown in Table 5.3).

5.3 Experimental Evaluation of Case Study I

In this section, we provide experimental results to demonstrate the impact of the proposed method on the efficiency of fault injection and fault analysis parts of the DFIA attack. In the injection part, an adversary applies the physical stress (e.g, clock glitches) and collects faulty outputs. In the analysis part, the collected faults are analyzed to retrieve the secret key. We attacked the TBOX AES program running on FPGA implementation of LEON3 for two cases:

1. **Grey-box approach:** In this case, the adversary uses the information provided by the instruction fault sensitivity model. The adversary also has the knowledge of the software behavior in the pipeline. Therefore, the adversary can identify the most suitable clock cycles and fault intensity range for fault injection, aiming at creating biased data faults.

2. **Black-box approach** In this case, the adversary does not use have the instruction fault sensitivity model. In addition, the adversary's knowledge about the pipeline behavior of the software is limited. The adversary can still do a limited timing characterization for the processor using the existing black-box approaches [14], [29], [13]. However, the pipeline state is unknown to the adversary. Therefore, the adversary cannot combine this timing characterization with the software behavior in the pipeline.

In both cases, our purpose was retrieving the AES secret key with the DFIA attack. We used one plaintext value and one key value for our experiments. We collected faulty ciphertexts for different fault intensity values. We controlled the fault intensity

Table 5.4: Comparison of Fault Injection Cost for Black-box and Grey-box Attack Strategies on TBOX AES

	Number of Attacked Cycles	**Tglitch Range**	**Number of Fault Injections (Step size = 162ps)**
Black-box Approach	16	[3.0, 15.8]	1280
Grey-box Approach	9	[5.43, 6.59]	81

by increasing/decreasing the glitch width T_{glitch}. Here, we present our results for retrieving one byte of the AES key.

The adversary starts with the fault injection part to collect biased data faults, which are required for the fault analysis part of the DFIA attack. In the grey-box strategy, the adversary uses MAFIA (Section 3) to find the most suitable points and injection parameters for biased data fault injection. Then, he injects faults into these points in the execution of AES software and collects faulty ciphertexts.

In the black-box case, the adversary has a limited knowledge of the target system. Therefore, to collect biased data faults, he has to exhaustively inject faults into all points in the execution of software and hope they will lead to biased data faults. However, in this approach, a large number of fault injection attempts will cause random effects in the fault injection point. For example, a fault injection attempt might affect the address calculation of a memory-load instruction and causes fetching an irrelevant data from the memory. This will create a random effect, rather than a biased effect, in the fault injection point. Although this fault injection creates a faulty ciphertext, it will not be useful for DFIA. Next, we investigate the required effort for biased fault injection for both cases.

Table 5.4 demonstrates the effect of proposed method on the efficiency of fault injec-

tion part of the attack. The table shows T_{glitch} range and the number of clock cycles that an adversary can exploit to create biased data faults. The first column of the table lists the number of clock cycles during which a fault injection attempt might yield a biased data fault. The total number of these cycles for the TBOX AES is 16 (5.3). For the black-box case, the adversary attempts to inject faults into all of these cycles as he is not aware of the pipeline behavior of the AES software. For the grey-box case, this number reduces to 9 with micro-architecture aware fault attack strategy.

The second column of Table 5.4 shows the T_{glitch} range in which the adversary attempts to inject biased data faults. The overall range for our LEON3 implementation and experimental setup is $[3.0, 15.8]$ns, as the critical path of the processor is 15.8ns and the minimum glitch period of our setup is 3.0ns. For the black-box strategy, the adversary tries all possible glitch width (i.e, fault intensity) values. For the grey-box case, this range will be reduced due to the proposed methodology. For every target cycle, we will have a different valid T_{glitch} range as it is previously shown. In our case, there are 9 valid T_{glitch} ranges. In Table 5.4, we list the average lower and upper bound values for the grey-box approach rather than showing bounds of each $Tglitch$ range.

The third column of Table 5.4 is the total number of fault injection attempts for *162*ps step-size. These numbers show the required fault injection attempts to explore all possible T_{glitch} values that might yield biased data faults. We obtained these numbers by multiplying the number of attacked clock cycles and the number of T_{glitch} levels within the corresponding T_{glitch} range. As it can be seen, the number of total fault injections for the black-box case (1280) is greater than the grey-box case (81). As a result, the proposed methodology significantly increases the efficiency of

the fault injection part by reducing the effort for biased fault injection.

In the fault analysis part, the adversary uses the collected faults to retrieve the secret key byte. In this experiment, the DFIA attack was able to retrieve the key byte with 31 faults in the black-box case, and with 17 fault injections in the grey-box case. In other words, the faults in the grey-box case provide more information on the secret key then the faults in the black-box case do. This is expected as the proposed method increases the control of the adversary on the induced fault effects.

In conclusion, these experimental results demonstrate that DFIA attacks are feasible on pipelined RISC processors for the black-box and grey-box approaches. They also show that the grey-box strategy significantly reduces the fault injection effort required to create biased data faults, and it enables more efficient DFIA attacks.

5.4 Experimental Evaluation of Case Study II

For each attack scenario and countermeasure investigated in Section 5, we launched a clock glitch injection campaign using our experimental setup. In the following paragraphs, we will list the observed faulty behavior for each countermeasure.

Table 5.5 shows the results of our experiments on the **Instruction Duplication** countermeasure. The first column of this table, shows the fault injection scenario. Column 2 and 3 show the clock cycle for the fault injection and the glitch width, respectively. The last two columns list the faulty instructions and the observed fault effect.

In our experiments, we were not able to achieve the *Scenario A.1.*, which requires injecting the same fault into both copies of the instruction with a single glitch in-

Table 5.5: Fault Injection Results on Instruction Duplication Countermeasure

Glitching Scenario	Target of Fault Injection	Glitch Width (ns)	Impacted Instruction(s)	Observed Fault Effect
A.1. Single Glitch	-	-	-	-
A.2. Single Glitch	*Cycle 2*	9.0 - 12.6	LD1, A CMP, F	Fault in `%g2` `CMP` to `SRL`
A.2. Single Glitch	*Cycle 4*	12.0 - 14.6	LD1, M BNE, F	Fault in `%g2` `BNE` to `NOP`

jection. We observed that the effect of fault on the two `LD` instructions is different because they are in different stages of the pipeline. The table shows that we have successfully injected faults that result in other scenarios. We successfully created *Scenario A.2.* by injecting a single glitch fault into *Cycle 2* or *Cycle 4*. By injecting glitches in *Cycle 2*, we were able to affect the operands fetched in the *Stage A* of `LD1` instruction, and change the value stored in `%g2`. This fault injection also affected the *Stage F* of `CMP`, and changed this `CMP` instruction into a shift-right instruction (`SRL`). By injecting a fault in *Cycle 4*, we affected the *Stage M* of `LD1` and *Stage F*. This fault injection caused a faulty value in the result of `LD1` (`%g2`), and replaced the branch-on-not-equal (`BNE`) instruction into a `NOP` instruction. Therefore, the code did not jump to the error handling procedure although there was a mismatch between the results of `LD1` and `LD2`.

Table 5.6 shows the result of fault injection into the **Instruction Parity** countermeasure. As shown, since the parity countermeasure has many instructions for computation of the parity bit, there are several points in the program that are vulnerable to the fault injection. In this table, we show that the adversary can exploit single glitch injection to either corrupt multiple bits in the protected register's value or the computation of the parity bit. For example, by injecting the glitch in *Cycle 13*, the condition codes and the `XOR` instruction changes to a `NOP` instruction.

Table 5.6: Fault Injection Results on Instruction Parity Countermeasure

Glitching Scenario	**Target of Fault Injection**	**Glitch Width (ns)**	**Impacted Instruction(s)**	**Observed Fault Effect**
C.1. Single Glitch	*Cycle 4*	13.2	LD1, M	Fault in `%g3`
C.2. Single Glitch	*Cycle 13*	12.1- 13.3	LD1,W XOR2, E	Fault in `%g3` `XOR` to `NOP`

Table 5.7: Fault Injection Results on Instruction Skip Countermeasure

Glitch Scenario	**Target of Fault Injection**	**Glitch Width (ns)**	**Impacted Instruction(s)**	**Observed Fault Effect**
D.1. Single Glitch	*Cycle 1*	9.0 - 9.2	LD1, D LD2, F	`LD1` to `SETHI` `LD2` to `NOP`

Table 5.7 shows the result for the **Instruction Skip** countermeasure. As shown, with single, we are able to change the opcode of the `LD` instructions to other opcodes and skip two consecutive instructions. For example, by injecting fault in the *Cycle 1*, we converted `LD1` instruction into a `SETHI` instruction and `LD2` instruction into a `TADCC` instruction. The `SETHI` instruction uses the least significant 22 bits of the instruction value, and writes this value into the destination register (`%g2` in this case). `TADCC` is a special addition instruction and it does not update any register; it is effectively a `NOP`. As a result, we update the destination register with a random value. Other scenarios of fault injection are explained in [11].

Chapter 6

Micro-architectural Embedded System Simulator for Fault Injection (MESS)

Disclaimer: This chapter is an excerpt from a research internship project that was done by the author of this dissertation at Riscure. The views expressed in this chapter are those of the author and do not reflect the official policy or position of Riscure.

The previous chapters demonstrates that considering multiple layers together makes fault attacks better understood, and thus, it increases the efficiency of fault models and fault attacks. This chapter demonstrates a fault attack simulator, Micro-architectural Embedded System Simulator (MESS), which enables software developers to represent this better understanding of fault attacks in the simulation environment and assess vulnerability of the software programs against fault attacks.

The next section provides an overview of MESS. Section 6.2 gives implementation details of MESS. Section 6.3 presents the operation flow of MESS. Section 6.4 demonstrates different fault injection experiments using FAME. Finally, Section 6.5 discusses the related work.

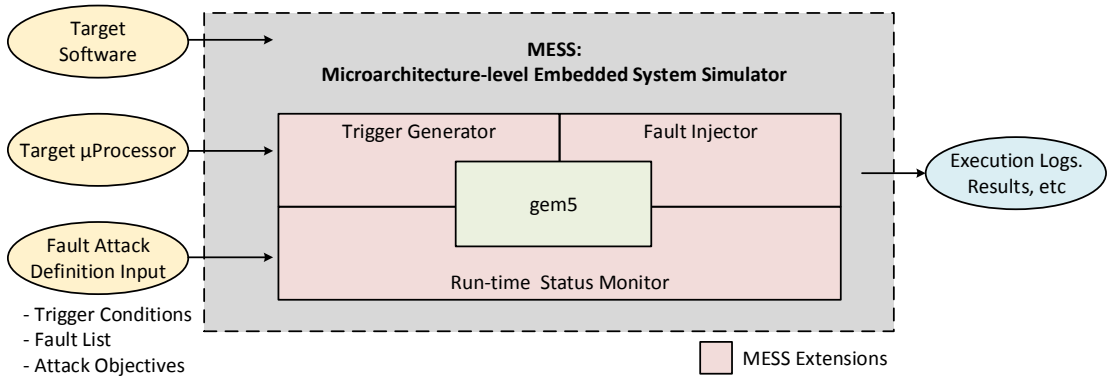


Figure 6.1: Block Diagram of the proposed fault attack simulator MESS: Inputs are target binary program, micro-architectural model of the processor, and definitions of the fault attack experiment. Output is a list of fault injection results.

6.1 Overview of MESS

MESS is based on an open-source, cycle-accurate, micro-architecture level, full-system simulator gem5 [74]. As it is shown in Figure 6.1, additional fault injection and analysis capabilities are integrated into gem5 to extend it to MESS. The extensions consist of three main components: Trigger Generator, Fault Injector, and Run-time Status Monitor. The Trigger Generator enables a user to define trigger conditions to enable/disable the fault injection. The Fault Injector enables a user to define timing, location, and type of the faults to be injected. It also performs the fault injection based on the defined fault parameters and trigger conditions. The Run-time Status Monitor allows a user to define run-time events to be monitored during the simulation. These events may be used to check if an attack is successful (e.g, bypassing a security check), or to get diagnostic information about fault behavior of the target program.

The inputs of MESS are an executable binary file for the target software, a micro-

architectural model of the target processor hardware, and the definition of the fault attack (trigger conditions, fault list, and run-time events). MESS will carry out the defined fault attack experiment, collect the results, and classify the collected results. If it is required, the user can also carry out further analysis on the interesting results by using debugging options of gem5.

The original gem5 simulator allows a user to define cycle-accurate models for main micro-architectural components and instruction execution. This enables MESS to close the abstraction gap between the view of a software designer and view of an attacker. A software designer develops the source code of a software program in a high-level language or in the assembly-level while a fault attack adversary injects the faults at the lowest level, during the execution of the program on the processor hardware:

- From the designer's perspective, a software program is a sequence of instructions that will be executed on the processor hardware.
- At the instruction-set architecture (ISA) level, each instruction is composed of several instruction-steps.
- On the processor, each instruction goes through the instruction-cycle (Fetch-Decode-Execute Cycle): The processor loads each instruction from program memory (instruction-fetch, F), then determines the meaning of the instruction through its opcode (instruction-decode, D), then executes the instruction (instruction-execution, E). The execution step can also be divided into sub steps such as accessing register file to read operands (register access, A), carrying out the required computation on a functional unit (computation, C), accessing the data memory if it is required (memory access, M), and updating

the register with the result of the instruction (write-back, W).

- The actual scheduling of the instruction-steps into clock cycles depends on the organization of the microprocessor hardware, availability of the required micro-architectural components, and dependencies between the instructions. Therefore, the attacker sees a program as a sequence of clock cycles, during which instruction-steps are running on the available hardware blocks.
- In a fault attack, fault injection is active during specific clock cycle(s), and it potentially affects the instruction-steps running during this active time period.

As a result, software designers are able to evaluate their programs against various realistic fault attack scenarios using cycle-accurate modeling capability of MESS. The next section details the components of MESS.

6.2 Components of MESS

This section explains the details of the main components of MESS that are gem5 simulator, Trigger Generator, Fault Injector, and Run-time Status Monitor.

6.2.1 gem5 Simulator

gem5 is a popular open-source system simulator [74]. It provides a modular platform for computer system-level architecture. In gem5, a user can define the main micro-architectural components (CPUs, caches, interconnects, memories, etc.) of a computer system in a cycle-accurate way. Then the user can run an application on this model and analyze its execution.

Supported ISAs: gem5 supports a number of ISAs, including Alpha, MIPS, ARM, Power, SPARC and x86. The simulators modularity allows these different ISAs to be easily implemented on top of the generic CPU models and the memory system. MESS supports ARM and x86 instruction sets.

Operation Modes: gem5 can operate in two modes: System Call Emulation (SE) and Full System (FS). In SE mode applications execute on simulated bare metal. In the FS mode, the user is able to run an operating system (OS) on top of the simulator. In this mode, applications are executed under the control of the OS. MESS supports both of the modes.

Debug Features: gem5 also provides several debugging options to examine the execution of the programs. It provides debugging flags to monitor the activity of the different parts of the system. It is also possible to connect the modeled system with gdb debugger.

6.2.2 Trigger Generator of MESS

In a fault attack, the fault injection needs to be active only for a specific time period during the execution the target program. MESS uses a simulator-level, global variable `isFaultEnabled` to indicate whether fault injection is active or not. Fault injection is enabled as long as the value of `isFaultEnabled` is 1. MESS controls the value of `isFaultEnabled` with a program counter (PC) based triggering mechanism, which enables or disables fault injection based on a list of user-defined trigger conditions. Using this mechanism, a user can enable and disable the fault injection during

Listing 1 Operation of The Trigger Generator

```
1: Input: User-defined list of trigger conditions chronological_trigger_list

   ▷ At the beginning of the simulation, MESS parses the input file and creates the
   queue of trigger conditions
2: triggerQueue ← PARSE(trigger_list)

   ▷ At each clock cycle, MESS checks and updates the trigger conditions
3: function UPDATE_TRIGGER_CONDITIONS
4:     tCond ← triggerQueue.head          ▷ The first trigger condition in the queue
5:     if PC == tCond.trigPC then         ▷ If the target PC address is encountered
6:         if tCond.trigCnt == 0 then                          ▷ If the condition is met
7:             isFaultEnabled ← tCond.trigIsSet  ▷ Enable/Disable fault injection
8:             REMOVE_HEAD_ELEMENT(triggerQueue) ▷ Remove the head element
9:         else
10:            tCond.trigCnt − −
11:        end if
12:    end if
13: end function
```

different parts of the target programs execution.

The user can specify a trigger condition using with 3 parameters, `trigPC`, `trigCnt`, and `trigIsSet`:

- `trigPC` is the program counter (PC) value associated with the trigger condition. During the simulation, the trigger generator continuously compares the PC of the simulated processor with the value of `trigPC`.
- `trigCnt` is the match counter associated with the trigger condition. The trigger condition is satisfied, when the processor's PC and `trigPC` match `trigCnt` times.
- `trigIsSet` is a boolean flag that determines if the fault injection will be enabled

or disabled after the trigger condition is satisfied. If the value of `trigIsSet` is 1, fault injection is enabled after the trigger condition is satisfied. Otherwise, the fault injection will be disabled.

The user feeds the trigger conditions to MESS in the form of a text file (`trigger_list`). Each line of the file corresponds to an individual trigger condition written in the following format:

```
<trigPC> <trigCnt> <trigID> <trigIsSet>
```

At the beginning of the simulation, MESS parses input file and puts the trigger conditions into a queue of trigger conditions, `triggerQueue`. Then MESS consumes each element of the `triggerQueue` one by one as it is shown in the Listing 1. At each clock cycle, MESS calls the function `update_trigger_conditions()` to check whether the first (i.e, head) trigger condition of the queue is satisfied. If it is satisfied, MESS enables/disables the fault injection based on the value of the corresponding `trigIsSet`, and it removes the trigger condition from the queue. MESS repeats these steps until there remains no trigger condition in the `triggerQueue`.

6.2.3 Fault Injector of MESS

The Fault Injector provides a mechanism to define and inject faults. gem5 provides a micro-architecture level, cycle-accurate model of the target programs execution. Therefore, MESS can inject cycle-accurate faults into main micro-architectural components of the system. These components include different pipeline stages, caches, memories, register file, branch predictor, and so forth. Moreover, MESS allows the implementation of both high-level (e.g, instruction-skip) and low-level (bit-flip) fault

models. For this work, we implemented a subset of the various potential fault models. Other fault models can be easily implemented using a similar approach.

The user describes the faults to be injected by feeding a list of faults into MESS in the form of a text file. Each line of the input text file describes a specific fault with 4 attributes: *Fault Location*, *Fault Timing*, *Fault Type*, and *Fault Duration*. The following paragraphs explain the fault injection capabilities of MESS for each of the attributes.

Fault Location: The fault location specifies the target micro-architectural component for fault injection. The current version of MESS supports fault injection into instruction fetcher, instruction decoder, read/write access to register file, ALU operations, read/write access to data memory, and condition flags.

Fault Timing: Fault timing attribute specifies the timing of the fault injection relative to the beginning of the simulation inside a fault injection window. It specifies the offset of the fault injection from the start of the simulation. A user can specify this offset in terms of executed clock cycles or executed instructions. Thus, MESS is able to inject faults into a specific instruction or a specific clock cycle.

MESS employs simulator-level, global fault-related performance counters to keep track of the timing information for the simulation. In the current version, there are five performance counters that count the number of fetched instructions, the number of decoded instructions, the number of executed instructions, the number of memory transfer instructions, and the number of clock cycles. At the beginning of the simulation, these counters are initialized to zero. Then, at each clock cycle of the simulation, they are updated based on the execution of the target program.

Fault Type: Fault type attribute specifies the fault behavior at the target location. The current version of MESS is able to flip a single bit of the target, XOR the target with a user specified mask (i.e, multiple bit flips), set/reset all bits of the target, assign target to a user-specified value, and invert the condition flags of the processor. Using these fault types, other fault types can also be emulated. For instance, MESS has two high-level fault types in addition to the bit-level faults. The instruction-skip type replaces the opcode of an instruction with the opcode of a NOP instruction during the instruction fetch. The conditional test inversion fault invert the result of a conditional test.

Fault Duration: Fault duration attribute specifies how long a fault will be active. Depending on the Fault Timing attribute, it can be specified in terms of number of clock cycles or number of instructions. For instance, we can flip a specific bit of the fetched instruction code for 3 consecutive cycles. It is useful to simulate transient, intermittent, and permanent faults.

At the beginning of the simulation, MESS parses the input file provided by the user, and creates fault queues. For each fault location listed in Table 6.1, a separate queue is created. MESS puts each fault described into an appropriate fault queue, in which faults are sorted based on their Fault Timing attribute. If fault injection is enabled during a clock cycle, MESS scans all of the fault queues to check if there is any fault scheduled for that clock cycle. If MESS finds a fault, it injects the fault and removes it from the queue.

Table 6.1: Fault Injection Capabilities of MESS. Each fault is defined by four attributes: Fault Location, Timing, Type, and Duration.

Fault Location	**Fault Timing**	**Fault Type**	**Fault Duration**
Instruction Fetch Instruction Decode Register File Access Memory Access Address Calculation ALU Computations Conditional Flags	At a specific instruction step At a specific clock cycle	Multi/Single Bit-flips Set All Bits Clear All Bits Assign Arbitrary Value Instruction Skip Conditional Test Inversion	Number of Clock Cycles Number of Instructions

6.2.4 Run-time Status Monitor of MESS

The run-time status monitor of MESS continuously observes the execution of the target program for catching run-time events that are important for the fault attack experiment. For instance, a run-time event can be to reach a specific iteration of a loop, to access a specific instruction, or to write a specific value into a register. A user may want to observe run-time events for various purposes:

- To check if the attack's objective has been achieved after fault injection,
- To check if the attack's objective has been failed after fault injection,
- To check if the simulation should be terminated,
- To collect diagnostic information about the fault behavior of the target program,

The run-time status monitor enables the user to define an action to be taken whenever a run-time event occurs. For instance, this action may be terminating the simulation and printing an exit message.

The user specifies 3 parameters (`runID`, `runFlag`, and `runCnt`) to define a run-time event to be observed:

- `runID` is the unique identification number of the run-time event.
- `runCnt` determines how many times the observed event needs to happen before setting `runFlag`.
- `runFlag` is a boolean value associated with the run-time event. Its initial value is zero, and it is set to 1 if the run-time event occurs `runCnt` times.

For flexible handling of the run-time events to be observed, MESS allows the user to modify a small part of its source code via a set of Application Programming Interface (API) functions. The API functions are written in C++, and they are explained below:

- `run_declare()`: In this function, the user declares each run-time event as an instance of the class `RunTimeEvent`. During the declaration, the user needs to specify the values of `runFlag` and `runCnt`. The following code listing shows an example template for run_declare() function. In this example, three run-time events are declared.

```
void runDeclare() {
   //new RunTimeEvent(<runID>, <runCnt>);
   new RunTimeEvent(1, 1); // Success
   new RunTimeEvent(2, 6); // Exit
   new RunTimeEvent(3, 3); // Observation
}
```

- `run_update(RunTimeEvent rEvent)`: This function checks if a given run-time event `rEvent` has occurred. Based on the result of the test, it updates the status flag `rEvent.runFlag` and the counter `rEvent.runFlag`.

  The following code listing demonstrates a template. The body of the given template is organized as a `switch` statement that switches on `rEvent.runID`. This `switch` statement includes a separate `case` declaration for each of the observed events. The template shows an example user code for the first event, which occurs whenever program counter is `0x8A5C`. The code first checks if the event has been reached, and then, it updates the `runCnt` value accordingly. The run-time status of information of the system is accessed using gem5 infrastructure such as the function `gem5.getReg()`. Finally, the `if` statement at the end of the function raises the status flag `runFlag`. This structure provides flexibility for the user.

```
void runUpdate(RunTimeEvent* rEvent) {
   // Update runCnt value
   switch(rEvent.runID) {
      case 1 : if((gem5.getReg(PC) == 0x8A5C)) // Check
         Event
               if (rEvent.runCnt > 0) // Update counter
                  rEvent.runCnt --; break;
      case 2 : ...; break
      case 3 : ...; break;
   }

   // Update runFlag Value
   if (rEvent.runCnt == 0x0)
```

```
        rEvent.runFlag = 1;
}
```

- `run_flag_handler(bool* flags)`: The input of this function is the array of status flags of the declared events in `run_declare()`. The user describes the actions to be taken based on the values of individual flags or combination of flags.

  In the given example below, the function is used for terminating the simulation and printing an exit message that contains the values of all flags and whether the attack was successful.

```
void runFlagHandler(bool* flags) {
    // Success Message
    sprintf(message1, "SUCCESS, flags=%d%d%d",
  flags[1],flags[2],flags[3]);

   // Fail Message
   sprintf(message2, "FAIL, flags=%d%d%d",
                flags[1],flags[2],flags[3]);

    // Combining Flags
   if (flags[1] && flag[2])
      exitSim(message1);
   else if (flag[2])
      exitSim(message2);
}
```

Listing 2 Operation of The Run-time Status Monitor

```
    ▷ At each clock cycle, MESS checks and updates the run-time events
 1: function UPDATE_RUNTIME_EVENTS
 2:     for each run-time event rEvent do
 3:         RUN_UPDATE_(rEvent)                          ▷ Update runCnt and runFlag
 4:         statusFlagArray[rEvent.runID] ← rEvent.runFlag    ▷ Array of flags
 5:     end for
 6:     rCond ← triggerQueue.head          ▷ The first trigger condition in the queue
 7:     RUN_FLAG_HANDLER(statusFlagArray)                ▷ Handle status flags
 8: end function
```

After completing the code for declaring and handling the run-time events, the user needs to recompile MESS. During the run-time, MESS uses the API functions to check and update the status of the run-time events as it is shown in Listing 2.

6.2.5 Cycle-wise Operation of MESS

This section briefly describes the operation of MESS for each cycle (Listing 3). For each cycle, MESS first checks the run-time events and trigger conditions. If any event has occurred or any trigger condition has been satisfied, MESS takes the corresponding action. This action can be enabling/disabling the fault injection, terminating simulation, or setting a status flag depending on the type of the checked condition. Then, if the fault injection is enabled, MESS scans the fault location queues to check if there is any fault scheduled for this specific clock cycle. If a scheduled fault is found, the fault location is modified in accordance with the type of the fault.

Listing 3 Cycle-wise Operation of MESS

```
1: for each clock cycle clk do
2:     UPDATE_RUNTIME_EVENTS( )                          ▷ Listing 2
3:     UPDATE_TRIGGER_CONDITIONS( )                      ▷ Listing 1
4:     if isFaultEnabled then
5:         scheduledFaults ← SCAN_FAULT_QUEUES(clk)
6:         for each scheduledFault f do
7:             INJECT_FAULT(f)
8:         end for
9:     end if
10: end for
```

6.3 Designing and Running Experiments on MESS

Figure 6.2 shows the main steps to design and run a single fault injection experiment on MESS. The design of a fault attack experiment starts with preparing an executable file for the target software and a gem5 model for the target hardware. Second, the user analyzes the target program to determine attack objectives, trigger conditions to enable/disable fault injection, and run-time conditions to be monitored. Third, the user modifies the source code of MESS to integrate the run-time events, and compiles the source code. In the final step of the design phase, the user prepares text files that contain trigger conditions and faults to be injected. To run the fault attack experiment, the user executes the compiled MESS executable file with the prepared input files. Finally, the user collects and analyzes the results of the experiment.

Preparing Target Software and Hardware (Step 1): MESS uses gem5 infrastructure to define the target hardware to be simulated. The components of the hardware and their interconnections are defined in the gem5 configuration script, which is written in Python. MESS needs an executable file as the target software.

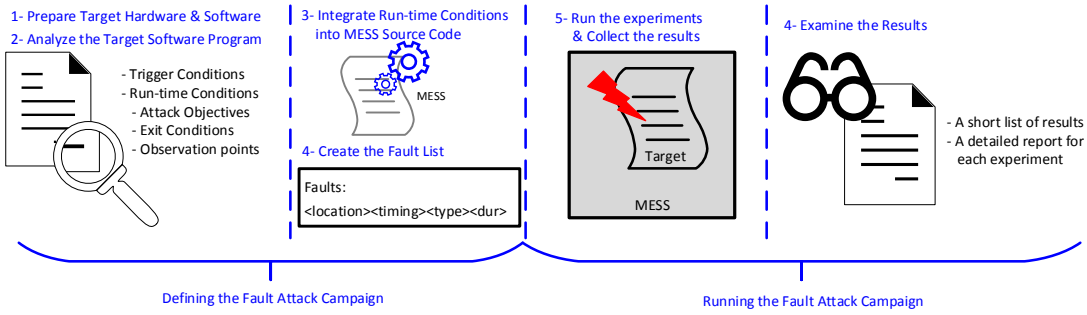


Figure 6.2: A typical flow for running a fault attack experiment on MESS

Therefore, the source code of the target software needs to be compiled and linked. If MESS works in the bare-metal (SE) mode, the target program must be statically linked. In full system (FS) mode, the simulated operating system takes care of any dynamic linking.

Analyzing the Target Software (Step 2): The user needs to analyze the compiled and linked executable of the target software to determine the objectives of the attack and run-time events to be observed during the simulation. These generally include checking if a specific program counter value is reached, if a specific register/memory location is corrupted, if control flow of the program is modified, or combination of them. The user also determines the parts of the code to be attacked. This is generally one-time analysis for a given software program. The user can use the results of this analysis for different fault attack experiments. For instance, the user may test an attack objective under several fault injection scenarios.

Integrating Run-time Conditions and Compiling MESS (Step 3): In this step, the user modifies the API functions of MESS to integrate the run-time events.

Then MESS is compiled using the infrastructure provided by gem5. This step is also one-time for a given target binary program and a given set of run-time conditions. The resulting binary file for the simulator can be used to test various fault injection cases.

Preparing the Text Files for Trigger Conditions and Faults (Step 4): In this step, the user prepares the list of trigger conditions and faults to be injected in the format of text files. Each line of the trigger condition file (*trigger_list.txt*) is in the following format:

```
<trigPC> <trigCnt> <trigID> <trigIsSet>
```

Each line of the fault list file (*fault_list.txt*) is in the following format:

```
<Fault Location> <Fault Timing> <Fault Type> <Fault Duration>
```

The user needs to be sure that fault injection hits the right point of the execution. MESS provides a command line option to assist the user in finding the appropriate `Fault Timing` values. When MESS is run with this command-line option, it prints the fault timing information for each instruction executed. The user can run MESS with this option before creating the fault list file.

Running the Experiment and Collecting the Results (Step 5): In this step, MESS is run using the the fault list (*fault_list.txt*) and trigger conditions list (*chronological_trigger_list.txt*) created in the previous step. MESS creates two output files in the text format. One of the files just contains the exit message defined in Run-time Status Monitor. The other file contains all of the simulation log printed

by gem5 infrastructure. More content can be included in the latter file by enabling debug flags of the baseline gem5 simulator [74].

Analyzing the Results (Step 6): In this step, the user analyzes the output files to check if the attack is successful as well as to get diagnostic information about faulty behavior of the target software.

6.4 Case Study: Fault Experiments on MESS

This section demonstrates the features of MESS by running different fault attack experiments on an example target program. The fault injection scenarios were selected such that they will represent practical fault injection experiments.

In these experiments, MESS is run on a VirtualBox VM with Ubuntu 14.4 operating system. The Virtual Machine is hosted by an Intel i7 41710HQ processor, which is a quad-core processor operating at 2.5 GHz frequency. During all of the experiments, MESS is run in its bare-metal (SysCall Emulation) mode. To automatically create and run multiple fault attack experiments, Bash scripts are used.

6.4.1 Target Hardware and Software

Figure 6.3a shows the source code of the target program. Using `memcmp()` function, the code compares two 20-byte buffers and returns `0x5A5A` or `0x00` based on the result of the comparison.

In our example, the buffers contain different data, and thus, the code returns `0x5A5A`

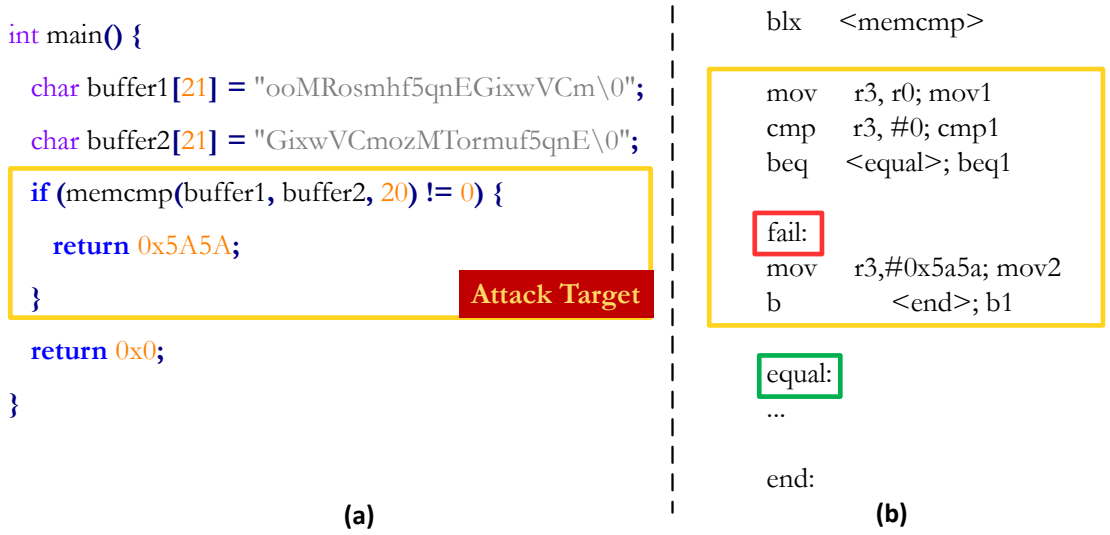


Figure 6.3: (a) Source code of the target program for the case study. The attack objective is bypassing the if statement and returning `0x00` as the result of the`main()`. (b) Disassembled target program for ARM32 (v7-A) architecture.

if there is no fault injection. The attack objective is altering the execution of the target program with fault injection such that the code returns `0x00` although the buffers contain different data. In Figure 6.3, we show the fault injection window with a yellow box.

Figure 6.3b shows the disassembled version of the target program for ARM32 (v7-A) architecture. The first three instructions inside the yellow box, compares the return value of the `memcmp()` with zero, which would be the return value if the buffers contained the same data. The code follows the `fail` branch if the `memcmp()` returns `0x00`; the `equal` branch is followed otherwise. In our example, the program will follow the `fail` branch because the `memcmp()` returns a value that is not equal to `0x00`. To make program to follow the `equal` branch, we will inject faults into the yellow box shown in Figure 6.3b. This part of the code will be called fault injection

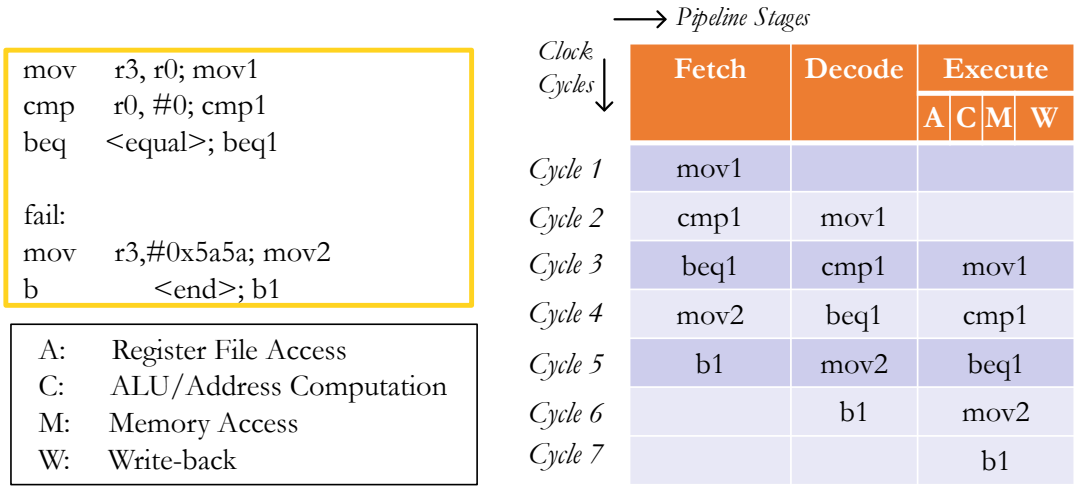


Figure 6.4: Execution of the fault injection window on the target harwdare model, an in-order processor with 3 pipeline stages.

window in the remaining part of this chapter.

For the experiments, we use an in-order, single-instruction-per-cycle processor model provided by gem5. The processor has three pipeline stages (Fetch, Decode, Execute). The processor completes multiple instruction-steps in the Execute stage in a single clock cycle: Accessing the Register File (A, ALU/Address Computation (C) Figure 6.4 shows the execution of the fault injection window on the target processor. As it is seen in Figure 6.4, this processor completes the execution of the fault injection window in 7 clock cycles. The next sections conduct different fault injection scenarios on the target software.

6.4.2 Attacking Individual Instruction Steps

Using MESS, a user can inject faults into individual instruction steps. This may represent precise laser shot injection, in which faults can be injected into a specific

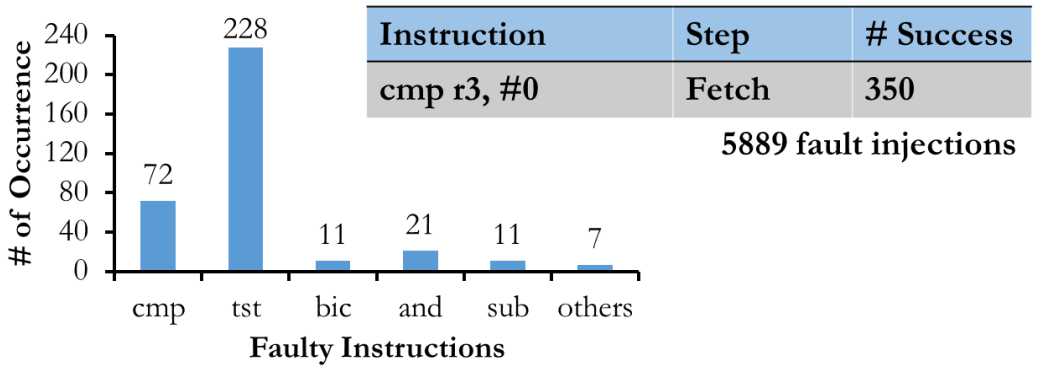
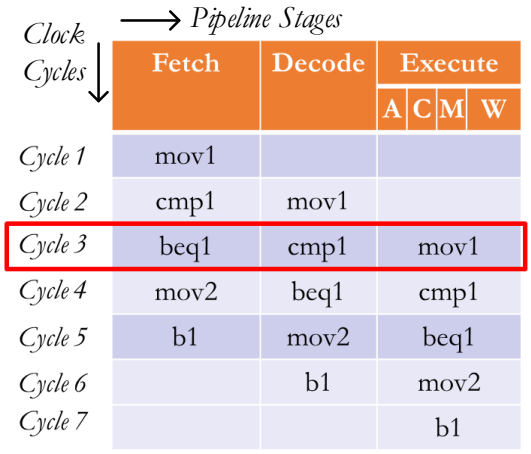


Figure 6.5: Results of attacking the fetch step of the `cmp` instruction.

part of the processor such as fetch unit.

In this case, we injected faults into fetch of the `cmp` instruction within the fault injection window (yellow box in Figure 6.4). During the fetch of the instruction from the instruction memory, we flipped up to three bits of the instruction. We exhaustively injected all of the 5889 possible faults.

Figure 6.5 shows the results of the experiment. As it is seen from the figure 350 of the 5889 fault injections successfully bypassed the security check. The graph in Figure 6.5 shows the distribution of those 350 instructions. As it is seen, the fault injection converted the `cmp` instruction into a `tst` instruction in most of the successful experiments. One reason of this behavior can be the value of the return value, which is `0x00`. A `tst` instruction applies an AND operation on its operands and update the condition flags of the processor in accordance with the result of the comparison. In our case, one of the operands is `0`. Therefore, the `tst` instruction sets the zero flag independent of the other operand. Another common pattern is converting the `cmp` instruction into a predicated instruction, which is conditionally executed depending



Clock Cycles	# Success
Cycle 2	208
Cycle 3	8
Cycle 4	51
Cycle 5	580
Cycle 6	0

8000 fault injections for each cycle

Figure 6.6: Results of attacking Individual Cycles (Cycle 2, 3, 4, 5, 6).

on the values of the condition flags. In our case, this effectively converts the `cmp` instruction into a NOP instruction.

6.4.3 Attacking a Single Clock Cycle

Using MESS, a user can also inject faults into individual clock cycle. This may correspond to clock glitch injection, in which a single clock cycle can be targeted. Other relevant real-world scenarios are voltage glitching, electromagnetic fault injection, or laser fault injection on a relatively slow target.

In this case, we injected faults into Cycles 2 to 6 of the fault injection window (Figure 6.6). For each cycle, we injected 8000 faults. We randomly selected the affected instruction steps among fetch, decode, and execution steps. For the fetch and execution stages, we flipped up to three bits of the fetched instruction and the computation result, respectively. For the decode stage, we replaced the index of

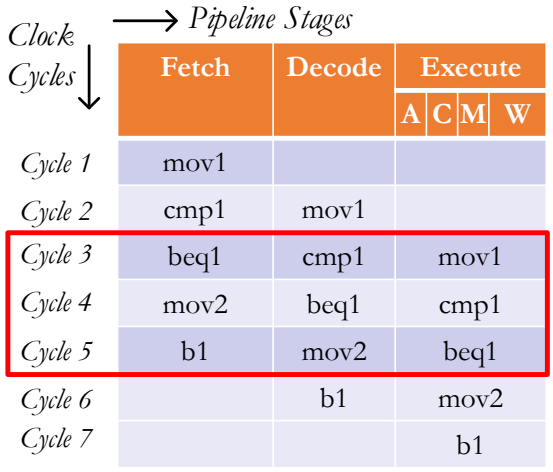
source or destination register of the target instruction with a random register index.

Figure 6.6 shows that the attack was successful for each clock cycle except Cycle 6. A possible reason of this behavior is the following. The branch instruction `b1` does not read any value from the register file. Therefore, our fault model does not have any impact on this instruction during the decode stage. In the execute stage, `mov2` instruction set the value of the register `r3` to `0x5A5A`. At the end of the `main()` function, the value of the `r3` is returned. As our fault model allows flipping up to 3 bits of the execution stages result, we cannot change `0x5a5a` into `0x00` with this fault model.

6.4.4 Attacking Multiple Clock Cycles

In the case of voltage, electromagnetic, or laser fault injection, especially for relatively fast targets, it is not always possible to affect only a single cycle of the execution. Rather, the fault injection might affect multiple consecutive cycles. MESS enables a user to simulate these cases as well.

In this case, we injected faults into 3 consecutive cycles within the fault injection window. We attacked the clock cycles 2-to-4, 3-to-5, 4-to-6, and 5-to-7. We injected 8000 faults for each group of the cycles. We used the same fault models as the fault models of 6.4.3. We selected the affected instruction steps and clock cycles randomly. Figure 6.7 shows the fault model and the obtained results. As it is seen, we successfully bypassed the security check for each case. The next section provides a brief comparison between MESS and the related work.



Clock Cycles	# Success
Cycles 2-to-4	194
Cycles 3-to-5	193
Cycles 4-to-6	199
Cycles 5-to-7	105

8000 fault injections for each row of the table

Figure 6.7: Results of attacking multiple clock cycles.

6.5 Comparing MESS with the Related Work

As the effects of faults on the electronic systems have been studied for more than 40 years for the reliability purposes, there is a good number of work on the fault injection simulators for both hardware and software implementations to assess their reliability [135, 136, 137, 138, 139, 140, 141]. However, the following points underline the differences between the faults in the reliability area and security area:

- In the reliability area, the source of the faults is random, natural events. In the security area, engineered, well targeted faults are injected by an adversary with malicious intentions.
- In the reliability area, most of the time, it is enough to model faults as single bit flips in the storage elements. It is also assumed that faults happen only once in a while. However, in the security area, an adversary can aim at affecting different

parts of the target system to achieve his/her malicious intent. The adversary is also capable of repeating the same faults and/or injecting a sequence of different faults. Therefore, it is not enough for security area to model the faults as single bit flips that occurs in once in a while.

As a result, a fault injection simulator targeting the security area has different, possibly more complex, requirements than a fault injection simulator targeting the reliability area. There are also research efforts from the security area to design a fault injection simulator [96, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152]. They provide source-code-level, assembly-level, and binary-level solutions. There are also emulation-based injection methods. However, these works have common limitations:

- Most of the proposed simulators focus on a separate level of abstraction. In contrast to MESS, they do not combine the effects of different abstraction levels on the fault behavior.
- Their fault injection is instruction-accurate and they completely abstract the hardware of the processor. However, processor hardware determines how a software program will be executed. In addition, in the actual fault injection, an adversary has a cycle-accurate view of the software execution. Therefore, a cycle-accurate simulator like MESS is a better solution for reflecting the actual fault occurrence.
  - With a cycle-accurate view, a user can see all of the instructions being executed on the processor for a given cycle. This gives an insight into which instructions will potentially be affected.
  - A cycle-accurate view can also provide a user to see the micro-architectural

effects such as data hazards, branch interlocks, and cache misses on the fault behavior.

- As they completely abstract the hardware structure, they have limited support for a user to specify which part of the processor will be affected by a fault injection. This can be a problem while modeling the effects of local fault injection methods such as laser and electromagnetic fault injection.
- Each of the previous work focuses on a very specific fault model. MESS provides a generic infrastructure to implement arbitrary fault models.

As it is previously discussed, the success of a fault attack on a software program depends on software program, instruction-set architecture, and the underlying processor. MESS simulates all of these abstraction layers together, and thus, enables a user to reflect attackers capabilities in the simulation.

Chapter 7

Fault-attack Aware Microprocessor Extensions (FAME)

This chapter presents the architectural details and the prototype design of the proposed technique, Fault-attack Aware Microprocessor Extensions (FAME), to detect and react to fault attacks on embedded software. FAME is a generic countermeasure, which combines a micro-architecture extension in hardware with a secure trap in software. The combined extension leads to fault-attack-resistant FAME processor with a secure exception mode to handle fault attacks. The microprocessor employs a low-level hardware checkpointing mechanism to recover from fault injection. A high-level secure trap in software then enables an application-specific response. The trap is user-defined and can be co-developed with the application. The combination of hardware fault detection and recovery, with a high-level fault response policy in software leads to significantly lower overhead when compared to traditional redundancy-based techniques in hardware or software.

The chapter is organized as follows. The next section describes the architectural and micro-architectural components of the fault-attack-resistant FAME processor. Section 7.2 summarizes the advantages of FAME. Section 7.3 lists the members of the implementation team of FAME prototype and their contributions to the imple-

mentation of the prototype. Section 7.4 presents the details of the chip prototype of FAME. Chapter 8 provides evaluation results for FAME.

7.1 Architectural Components of FAME

In this section, we describe the architectural design concepts in FAME. The main objective of FAME is providing minimum architectural support in hardware-level for enabling software programmers to minimize security risk resulting from fault injection. These extensions are generic, low-cost, and applicable to a broad class of embedded processor.

Figure 7.1 shows the overall architecture and operation of FAME. A fault injection attempt is detected in hardware by Fault Detection Unit (FDU). Fault response is achieved by a secure trap mechanism. Fault Control Unit (FCU) and Fault Response Registers (FRR) provide hardware support for this mechanism. Upon detection of a fault, FCU initiates the transition to a secure trap handler. The FRR contains the checkpoint information which enables recovery of the processor from fault injection. The secure trap handler applies a user-defined, application-specific security policy in a safe mode. Next, we will explain the details of FAME's components and operation.

7.1.1 Fault Detection

FAME relies on a hardware fault detection unit (FDU), which is a set of detectors monitoring processor's operation to detect anomalies. During the normal operation, an application runs in the nominal mode and no overhead is accrued. Upon detection of a fault, FDU asserts an *alarm* signal to notify the processor of a potential fault

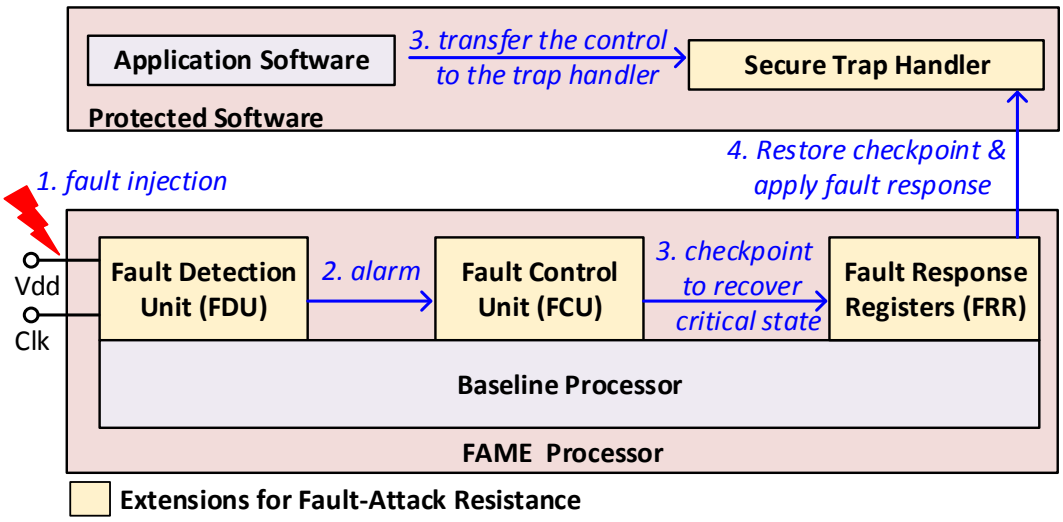


Figure 7.1: FAME combines hardware fault detection, hardware checkpointing, and software fault response.

attack.

The detector configuration and sensitivity level of FDU depend on the application domain and the attacker model. Based on the requirements, FDU is able to derive the fault status of the overall processor by combining different types and number of fault detectors. These include clock/voltage glitch detectors [153], electromagnetic pulse detectors [154], concurrent error detection methods [155], shadow latches [156], and error detection codes [157]. The requirement of the detectors employed in FDU is that they must detect fault injection before a faulty value is used by a subsequent instruction. In this work, we detect faults that originate from timing violations and localized electromagnetic pulses. Given the error detection mechanisms enumerated above, however, it is clear that FDU mechanism is generic and that it can handle a multitude of fault detection mechanisms.

7.1.2 Critical State Checkpointing

For fault-attack-resistant execution of the secure trap mechanism, FAME provides a hardware-level checkpointing support, the fault response registers (FRR). FRR maintain the critical system state needed to recover from the fault injection. They include the status register of the processor, the return address to the interrupted program, and the register file inputs of the writeback stage. FRR utilize double-buffer redundancy to ensure fault-free recovery data. Using the contents of FRR, the secure trap handler is able to restore the processor state back to the fault-free state just before the fault injection.

7.1.3 Fault Response

To handle the fault injection, FAME first applies hardware-level precautions to prevent fault effects from spreading further. Then, FAME initiates a software trap handler to apply a user-defined and application-specific fault response.

The hardware-level Fault Control Unit (FCU) manages the invocation and execution of the secure trap mechanism. It acknowledges the alarm signal of the FDU and takes immediate actions in the hardware level. FCU locks the fault recovery information into FRR, annuls the instructions being executed in the pipeline, and disables write operations into the register file as well as memory. This enables two essential capabilities.

1. The faulty parts of the software-visible state, which are contaminated before the alarm is raised, can be recovered by the trap handler.

2. No more faulty results will be committed to the architectural state of the

processor after the alarm is raised.

Meanwhile, FCU stops the execution of the user application, invokes a non-maskable Secure Trap Handler, and switches the processor from nominal mode to safe mode. This switching is done immediately at the next clock cycle. In safe mode, the processor is aware of the fault injection, and it can handle the faults through the secure trap handler. The Secure Trap Handler retrieves the fault-free processor state from FRR and then applies a user-defined fault response. The processor then returns to the instruction that was affected by the fault injection, and resumes execution from there. A side effect from this operation is that the processor eliminates fault-triggered instruction-skip.

It is mandatory that FCU responds to further fault injections and prevents fault attacks on the trap handling mechanism. In this work, FCU restarts the software trap handler if a fault injection is detected during safe mode. This guarantees that FAME cannot exit from safe mode without completing the trap handler.

7.1.4 Added Instructions

FAME adds two instructions to the baseline architecture to access FRR and to recover the fault-free state of the processor. The *Read FRR* instruction `RDFRR Rd,Rs` reads the content of the fault response register `Rs` into the architectural register `Rd`. Using this instruction, one can access the FRR content for the secure trap handler's return address and processor status register. The *Write FRR* instruction `WRFRR` restores the fault-free register file state from the corresponding FRR content.

The specific implementation of `RDFRR` and `WRFRR` instructions depends on the baseline

instruction set architecture. In this work, we use two existing instructions of the SPARC architecture as it is explained in Chapter 7.

In summary, FAME uses fault detectors that are combined into a processor-level alarm signal. The alarm signal initiates a software trap to decide on the further course of action. FAME provides hardware-level support to maintain the fault recovery information. It is up to the trap handler to decide if it is safe to continue execution or not. FAME ensures that the trap handling mechanism itself is protected from faults.

7.2 Advantages of FAME

The proposed extensions provide a low-cost, performance-efficient, adaptive, and backward-compatible mitigation of fault attack risk. Cost-efficiency is achieved by limiting hardware redundancy to a small subset of the processor state (fault control unit, fault recovery registers), and by limiting the software overhead to a small portion of the embedded software (secure trap handler). From a performance point-of-view, these extensions cause negligible timing overhead on the processor hardware because they work in parallel with the processor's original datapath. On the software side, the extensions affect the performance only if a fault is detected. The actual software overhead depends on the complexity of the trap handler, but will be smaller than the inherently-redundant software-only techniques. The secure trap mechanism supports a flexible fault response, and this provides several advantages.

- The secure trap is a uniform, generic mechanism to attach fault countermeasures. We will demonstrate this with several examples including AES encryp-

tion as well as to secure PIN code testing.

- The secure trap is flexible and allows an adaptive response to fault attacks. For obvious security concerns, we do not allow changing the trap handler at run-time. However, since the trap handler is aware about the interrupted program counter, it's possible to adjust the fault response depending on the application context (AES encryption versus PIN verification, for example).

- When working with multiple redundant fault sensors, the secure trap handler allows to combine the input of multiple sensors. This helps to distinguish a spurious fault activity from a genuine fault attack.

- From the developer's point of view, the secure trap handler implies that fault countermeasures can be designed separately from the secure application. This improves clarity and ease of use, when compared to traditional redundancy-based software application development.

By keeping fault response separate from application code, we also obtain backward-compatibility with existing cryptographic libraries and binary code. Furthermore, the proposed fault response strategy will also preserve the side-channel resistant properties of existing cryptographic applications. Indeed, composable countermeasures integrated in the application remain a challenging problem. For example, the cryptography library NaCl provides constant-time software implementation, but no fault-injection resistance [158]. Fault resistance can be achieved by adding redundancy in the NaCl source code, but this carries the risk that one destroys the constant-time properties.

7.3 Contributors to the FAME Prototype

Table 7.1: Contributors to the FAME Prototype

Team Member	**Contribution**
Bilgiday Yuce	▶ Architectural Design of FAME Processor and SoC ▶ RTL Coding of fault-attack-resistant features of FAME ▶ RTL Coding of fault-attack-analysis features of FAME ▶ RTL Coding and Testing of timing sensor (1^{st} tape-out) ▶ RTL Integration of Peripherals and Coprocessors to SoC ▶ RTL Simulation of Processor and Peripherals ▶ Gate-level Simulation of Processor and Peripherals ▶ RTL and Gate-level Fault Simulation of FAME processor ▶ Development of FAME-protected Software Programs ▶ Evaluation of of FAME-protected Software Programs ▶ Preparation of RAM macros for memories in the design
Chinmay Deshpande	▶ RTL Coding and Testing of timing and EM sensors ▶ RTL Coding and Testing of the Sensor coprocessor ▶ RTL Coding and Testing of the AES+ Coprocessor ▶ RTL Simulation of Processor and Peripherals ▶ Preparation of Standard-cell and I/O Pad Libraries ▶ Development of RTL to Gate-Level Synthesis Scripts ▶ Development of Scripts for Physical Implementation (Placement, Power Grid & Clock Tree Synthesis, Routing) ▶ Insertion of Scan-chain ▶ Development of a Test Board (FInalyzer)
Marjan Ghodrati	▶ Development of Scripts for Physical Implementation (Placement, Power Grid & Clock Tree Synthesis, Routing) ▶ DRC and LVS Check ▶ Utility Scripts for Dummy Poly, Metal and Via Insertion ▶ Final Signoff and GDSII Generation
Abhishek Bendre	▶ RTL Coding and Testing of AES Coprocessor ▶ Development of a Test Board (FInalyzer)
Mostafa Taha	▶ RTL Coding of Keymill-LR Coprocessor
Nahid F. Ghalaty	▶ Architectural Design of FRR
Conor Patrick	▶ Developing Code Templates for FAME Trap Handler

Previously described Fault-attack Aware Fault Extensions (FAME) transforms a baseline microprocessor into a fault-attack-resistant processor. To evaluate the performance and security of the proposed FAME extensions, we implemented and manufactured a chip prototype of the previously described FAME architecture in 6-metal-layer TSMC 180nm technology. The prototype is a System on Chip (SoC) containing the fault-attack-resistant FAME processor, several peripherals, and coprocessors. We have completed two chip tape-outs for the designed prototype. The first tape-out was completed on 10/31/2016, and the chips were returned from the fabrication and packaging on 01/13/2017. The second tape-out completed on 09/06/2017, and the chips were returned from the fabrication on 10/23/2017, and they were sent for packaging.

The FAME prototype design is the result of a collective effort of a design and implementation team, sponsored by the National Science Foundation Grant 1441710, and in part through the Semiconductor Research Corporation (SRC). The team is led by Dr. Patrick Schaumont and Dr. Leyla Nazhandali. Table 7.1 lists the student members of the team and their contributions to the development of the FAME prototype.

7.4 Chip Prototype of FAME

We designed the prototype as a System-on-Chip (SoC) architecture centered around a RISC processor with fully integrated fault-attack-resistant extensions, and the capability to execute a secure trap.

The prototype also contain a detailed fault-injection and debugging infrastructure

to assist the development and testing of fault attacks on the chip. Hence, the FAME prototype serves a dual purpose. First, it serves as a prototype for countermeasure design, and it supports the development and testing of secure trap handlers. Second, it serves as a platform to develop fault attacks and to study the fault response of the processor against fault injection. The detailed post-injection analysis features in the chip allow us to understand the precise impact of a fault injection.

This chapter provides implementation details of the FAME prototype, which was designed with a specific attack model in mind. Next, we specify the considered attacker model.

7.4.1 Attacker Model

In the design of FAME prototype, we assume the following attacker model. First, the attacker can observe and tamper chip input/output pins, clock pins and power/ground pins. For example, the attacker can mount clock-glitching and power-glitching attacks [14]. These attacks affect the entire chip network, and they can be caught with a single sensor. The attacker model also includes localized fault injection using electromagnetic pulses (EMFI) [13, 33]. However, we assume that the attacker cannot use high-cost laser fault injection that requires decapsulation of the chip package [10]. Third, similar to the attacker model of Lemke-Rust and Paar [159], we exclude invasive adversaries from our threat model. The attacker is not capable of directly modifying or monitoring the internals of chip. In addition, the attacker cannot modify software and firmware by any other mechanism except by injecting faults. The attacker thus cannot tamper with the secure trap handler, and she cannot overwrite on-chip memories. This means that we assume a protected

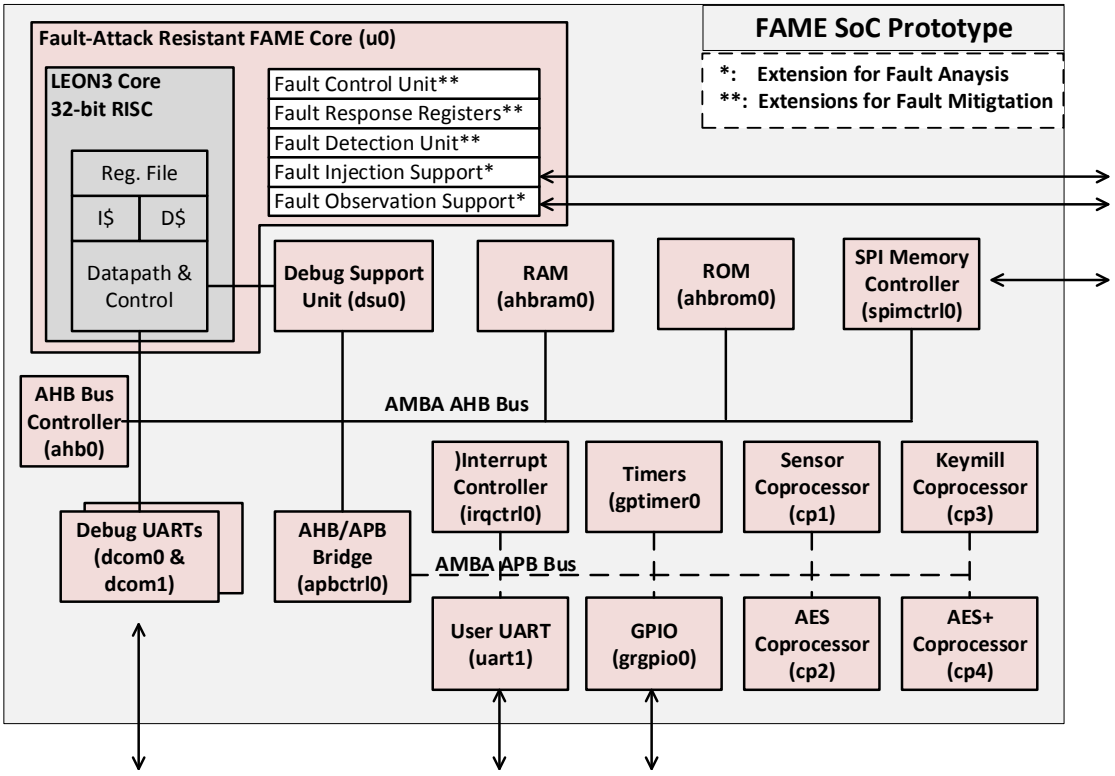


Figure 7.2: Block Diagram of the FAME SoC prototype. The FAME extensions are integrated into a LEON3 core. The chip includes on-chip memory and a collection of coprocessors used for fault sensing and cryptographic acceleration.

embedded-software execution environment, one which would be able to handle both the I/O attacker model and the memory attacker model as defined by Piessens [2].

7.4.2 Overall Design of FAME Prototype

Figure 7.2 shows a top-level block diagram for SoC architectures of the FAME chip. The components of the SoC are centered around the fault-attack-resistant FAME processor core, which is built by integrating FAME extensions into a 32-bit LEON3

core [121]. The FAME processor includes the previously described fault-attack-resistant extensions (i,e, FDU, FCU, and FRR) to detect and respond fault injection as well as chip debugging features (i.e, Fault injection and observation support) to aid the evaluation of fault effects on the chip. The fault-attack resistant extensions follow the attacker model described in Section 7.4.1. The chip debugging features on the other hand go beyond the permitted operations in the attacker model.

In addition to the fault-attack-resistant FAME core, FAME chip has several peripheral blocks that are interconnected through AMBA AHB and APB buses [160]. FAME chip includes the following peripherals, which are implemented as a part of an open-source SoC library GRLIB [125]. The details for their functional specification and implementation can be found in the user manual of GRLIB library [161]:

- A 64Kbyte on-chip RAM (`ahbram0`) to store the instructions and data of a program.
- A 1Kbyte on-chip ROM (`ahbrom0`) that sores a simple boot program.
- Two debug UARTs (`dcom0` and `dcom1`) to control the on-chip Debug Support Unit (`dsu0`).
- A user UART (`uart1`) for terminal I/O.
- A 4-bit GPIO (`grgpio0`), which can be used for integration of measurement equipment control with application software running on LEON3.
- An interrupt controller (`irqctrl0`) to manage the interrupt requests of the peripherals.

- A timer peripheral with five timer modules, which can be used for integration of measurement equipment control with application software running on LEON3. For example, the timers can be used for precise fault injection.
- A serial SPI interface access (`spimctrl0`) to an off-chip flash memory.

Moreover, the FAME prototype contains 33 timing sensors [75] and 160 EM sensors [162] distributed over the layout area, and 768 in-situ EM sensors integrated in the architecture state. The timing sensors are based on an internal, programmable delay line [75], while the EM sensors are based on a dual flip-flop configuration [162]. The FAME chip also contains the following custom-designed coprocessors:

- FAME has a sensor coprocessor (`cp1`) including 32 timing sensors and 160 EM sensors. The sensor coprocessor provides a logical view on the sensor configuration and supports a uniform software interface. In the chip back-end flow, a custom sensor-placement methodology was used to ensure the regular distribution of sensors over the chip surface.
- FAME also contains several hardware coprocessors, with and without in-situ fault detection. These coprocessors were added for experiments with fault injection in dedicated hardware structures. The coprocessors cover two cryptographic algorithms, AES and LR-Keymill [163]:
  - FAME has an AES coprocessor (`cp2`) with encryption/decryption functionality and support for Electronic Code Book (ECB) and Cipher Block Chaining (CBC) modes of operation. This coprocessor is an unprotected, 1 round-per-cycle design that serves as a reference hardware module for experiments with side-channel analysis and fault injection.

- FAME contains a protected AES coprocessor (`cp4`), derived from the other AES design, with in-situ sensor support. The internal registers used for encryption, decryption, and for the AES key-schedule are implemented using EM fault sensors, resulting in 768 in-situ EM sensors. In contrast to the generic sensors in the sensor coprocessor, the in-situ sensors do not require special placement, since they are intrinsically part of the protected design.
- FAME also includes a Keymill-LR coprocessor (`cp3`), a reference implementation of a leakage resilient keystream generator presented at the Hardware Oriented Security and Trust Symposium in May 2017 [163].

Next section presents the implementation details for fault-attack-resistant features of the FAME processor core.

7.4.3 Fault-attack-resistant FAME Core

FAME processor core consists of secure fault-attack resistant extensions integrated with an in-order RISC processor implementing the SPARC-V8 instruction set. The processor core is based on the open-source, synthesizable 32-*bit* LEON3 design[121]. The prototype chip includes a fault-attack resistant core with 2KB data cache, 1KB instruction cache and a register file.

Fig. 7.3 shows the 7-stage pipeline of LEON3 with FAME extensions. The pipeline consists of *fetch (F), decode (D), register access (A), execute (E), memory (M), exception (X),* and *write-back (W)* stages. FDU consists of sensors and generates and alarm signal. We integrated the FCU into the *X stage*. FRR provide fault

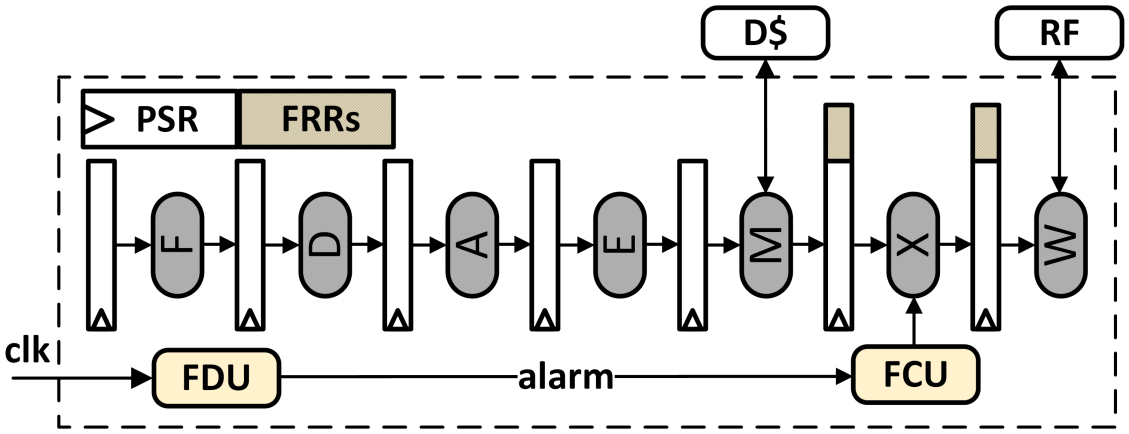


Figure 7.3: 7-stage LEON3 pipeline with FAME extensions

recovery information for some parts of the X and W stages because of two reasons. First, the Register File (RF) and Processor Status Register (PSR) are updated in the *W stage*. Second, the return address for the trap handler is computed in the *X stage*. The following subsections will discuss the features and design of various sub-blocks unique to the fault-countermeasure features.

Fault Injection to the Processor

To understand the operation of FAME, we need to carefully define the sequence of events leading to a fault. In a fault attack, an adversary waits until a program reaches a specific point in its execution. Then, he injects the fault into the program at this point. Finally, the adversary observes the fault effects after this point. In this work, we use *fault cycle* (C_f) to denote the clock cycle in which the fault occurs. We use *before-fault cycle* (C_b) and *after-fault cycle* (C_a) for the clock cycles before and after the fault, respectively. The program's execution is fault-free before C_f, and faulty from C_f. The embodiment of C_b, C_f, C_a depends on the fault injection method. Fig. 7.4 describes a generic situation as an example.

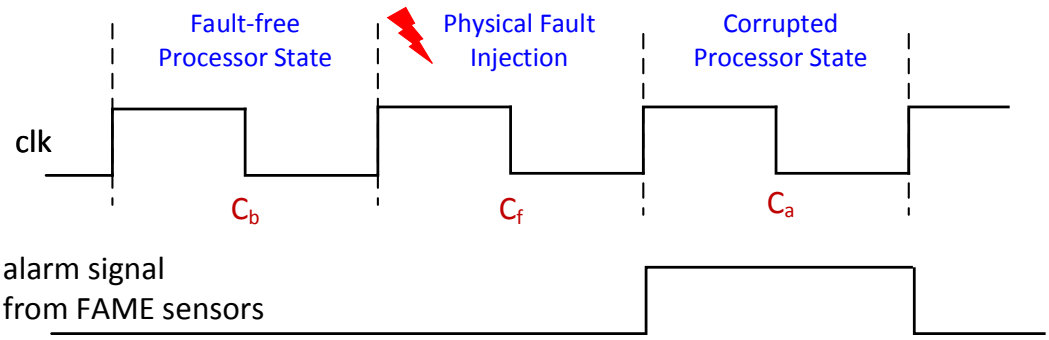


Figure 7.4: Fault Injection to the Processor: Adversary starts altering the physical operating conditions in cycle C_f. Before C_f, processor state is fault-free. The fault effects are captured into processor state during cycle C_f. FAME detectors generate an alarm signal just after C_f.

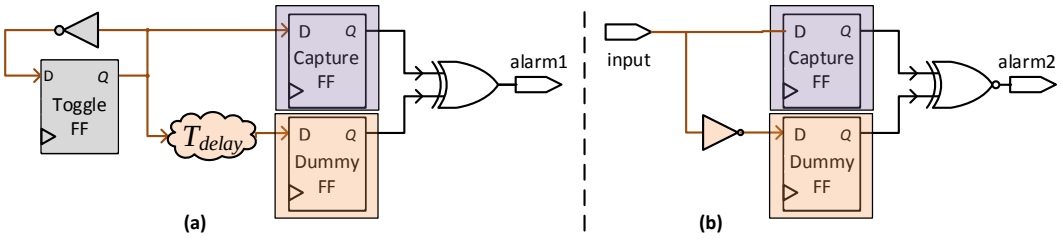


Figure 7.5: Block diagram of the sensors employed in FAME core: (a) Configurable delay-chain based timing attack sensor. (b) Dual complementary flip-flop based electromagnetic fault injection (EMFI) sensor.

FAME detectors are able to generate an alarm signal just after the fault effects are captured in the processor state. Next, we describe the structure and operation of sensors used in FAME prototype.

Fault Detection Unit (FDU)

In our prototype, Fault Detection Unit (FDU) employs two types of sensors to detect setup time violations and localized electromagnetic (EM) pulses.

Fault Detection Unit (FDU) detects setup time violation attacks with delay-based timing sensors similar to the implementations reported by [153, 154, 156]. Each of these sensors is made up of a delay chain of cascaded inverters (or fixed-delay digital circuit) designed such that its propagation delay is slightly greater than the critical path of the protected design. This ensures that the sensor is more sensitive than the slowest logic block. The reference delay value is fixed in the hardware logic and is calculated using static timing analysis. To overcome the process variation in the circuit and the sensor, we make the delay chain configurable. In addition to the delay-based sensor, FDU also employs another type of sensor, dual-complementary flip-flop based EM sensor [162], to detect localized electromagnetic (EM) pulses in addition to setup time violations.

Fig. 7.5a shows the implemented timing sensor, which consists of three FFs, a delay chain, an inverter, and an XOR gate. Toggle FF toggles its value every cycle which then arrives at Capture FF immediately and at Dummy FF after T_{delay}. In normal operation (T_{clk}), the inputs of both Capture and Dummy FFs toggle to the new value before the next clock edge. In case of a setup-time violation, Capture FF latches the new value whereas the Dummy FF latches the old value as the length of the clock period is not enough for delay chain to make transition to new value. Therefore, the XOR gate generates the alarm signal at the next upgoing clock edge.

Figure 7.5b demonstrates the gate-level structure of the dual-complementary flip-flop based EM sensor, which was proposed and demonstrated by Deshpande [162]. It employs two FFS, an inverter and an XNOR gate. The design of the sensor relies on the following observations from electromagnetic fault injection (EMFI) experiments on silicon [9], [164]:

- EMFI induces bit-set/bit-reset faults into a device through Eddy currents. The exact effect is determined by the polarity of EM pulse, which affects the direction of induced current flow.

- EMFI alters the value of a flip-flop by disrupting its switching process during the flip-flop is capturing a new value. This makes the flip-flop to capture a faulty value.

- For a given EM polarity, a fault injection can cause the value of a flip-flop to change either from logic-1 to logic-0 or from logic-1 to logic-0.

- The sequential elements on an integrated circuit is more vulnerable to EMFI than the combinational elements.

During the fault-free operation, the capture and dummy FFs of the sensor captures complementary values at each clock cycle. In case of EMFI, either the Dummy FF or Capture FF gets affected based on the polarity of the injected EM pulse. As a result of EMFI, the values of the Capture and Dummy FFs will be the same and the XNOR gate will raise an alarm signal. This sensor can be used either as a stand-alone sensor by feeding a constant input to it, or as an in-situ sensor by replacing original flip-flops of a circuit with the sensor.

FAME prototype employs 833 sensors in total to detect both setup-time violation attacks and localized EMFI. As it is shown in Figure 7.6, we logically organized the sensors and FDU of FAME into three blocks of the FAME SoC. The sensor coprocessor contains 32 timing sensors and 160 EM sensors. We integrated 768 in-situ EM sensors into the state of the AES+ coprocessor by replacing its original flip-flops with the EM sensors as explained in the Master Thesis of Chinmay Deshpande [162].

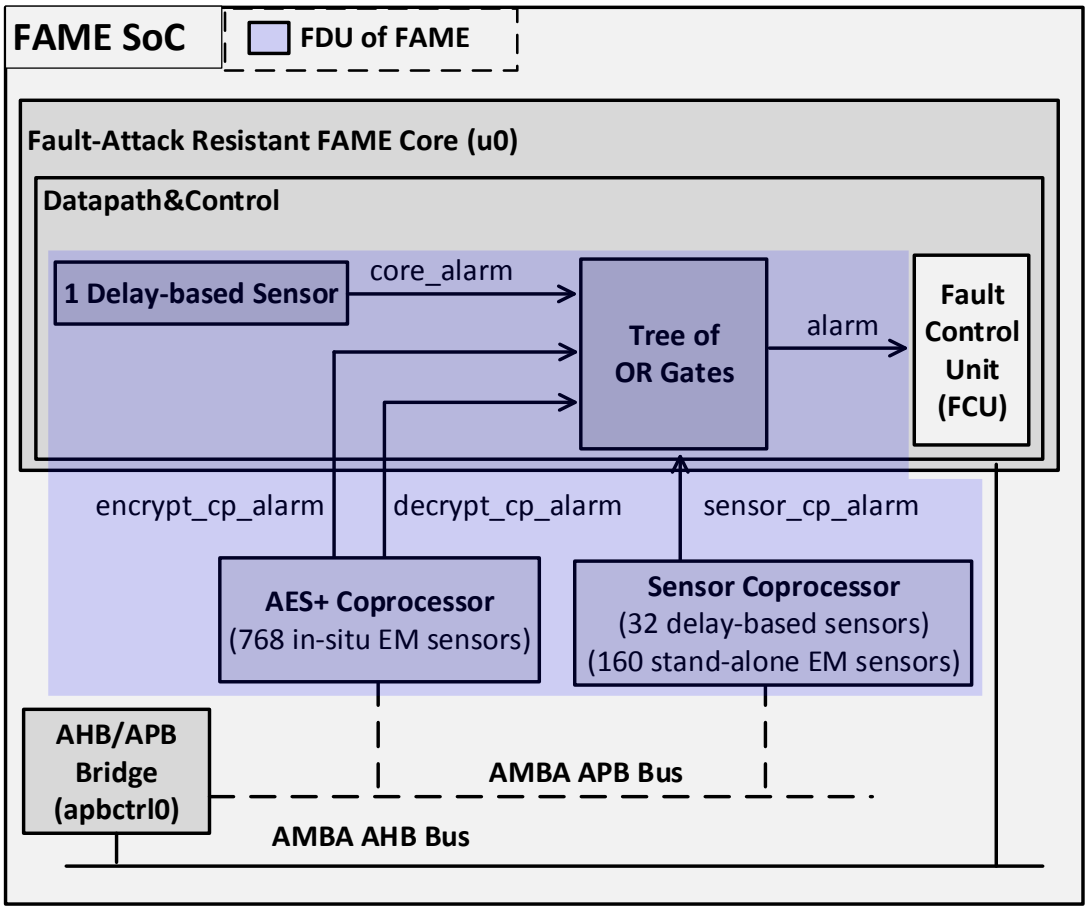


Figure 7.6: Logical Organization of the Fault Detection Unit

The datapath of the FAME core has a single timing sensor to detect global setup-time violation attacks.

The alarm outputs of sensors are combined via logical OR gates. The sensor coprocessor combines the output of individual timing sensors and EM sensors into a coprocessor-level `sensor_cp_alarm` signal. The AES+ coprocessor combines the output of its sensors into `decrypt_cp_alarm` and `decrypt_cp_alarm` signals, which

respectively corresponds to the sensors in the encryption state and the sensors in the decryption state. Finally, in the FAME core, FDU generates the processor-core-level `alarm` signal to trigger the Fault Control Unit (FCU) by combining `sensor_cp_alarm`, `decrypt_cp_alarm`, `decrypt_cp_alarm`, and the single delay-based sensor's output `core_alarm`.

From the physical point of view, the individual sensors are distributed over the chip. In the chip back-end flow of the sensor coprocessor, a custom sensor-placement methodology is used to ensure the regular distribution of sensors over the chip surface. This methodology was developed and implemented by Marjan Ghodrati, a student member of the FAME development team. In contrast to the generic sensors in the sensor coprocessor, the in-site sensors do not require special placement, since they are intrinsically part of the protected design. For fault analysis purposes, we also implemented a detailed configuration and status observation interface for the sensors.

The current prototype of FDU detects faults affecting the active instructions, including the opcodes and data. An adversary could also tamper with instructions and data at rest in the program memory or data memory. However, such tampering would not affect the software until these instructions are executed. Therefore, we can verify the application software against such faults using software-based error detection and correction codes [165], or by using checkpointing [52]. The integrity verification routines themselves are protected using the proposed fault mitigation methods in our prototype, or by software redundancy. Next, we will explain the implementation of FCU.

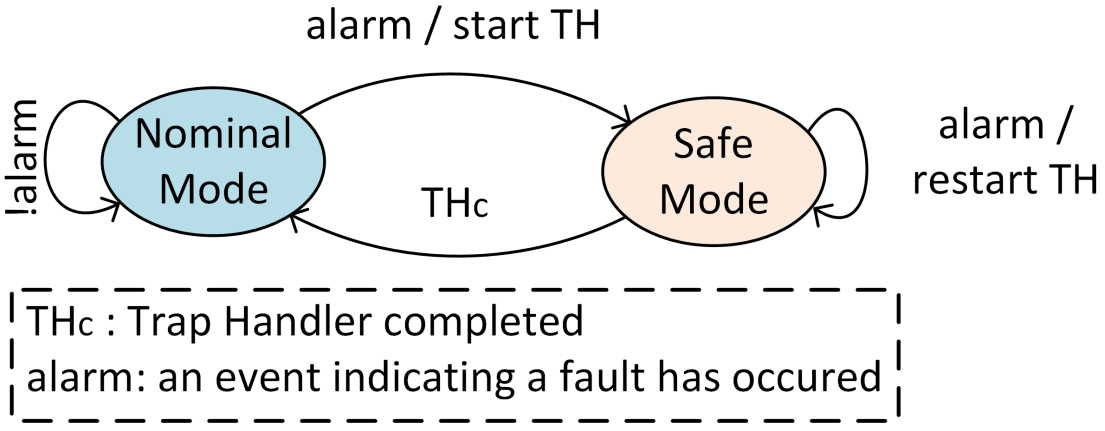


Figure 7.7: State diagram of the Fault Control Unit

Fault Control Unit (FCU)

FCU uses the state machine shown in Fig. 7.7 to manage the secure trap mechanism. If FCU detects an alarm signal while the processor is in the nominal mode, it switches the processor to safe mode. During this transition, the FCU annuls all instructions in the pipeline, disables all memory and register file transfers of the user application, locks the fault recovery information into FRR, and resumes execution with the first instruction of the trap handler. If the trap handler completes its execution without another fault attack detection, the FCU switches the processor back to the nominal mode. If the FDU detects a fault while the processor is in safe mode, the FCU restarts the trap handler and stays in safe mode. This guarantees that FAME cannot exit from safe mode without completing the user-defined security policy.

To initiate the fault processing after an alarm is raised, we extend the *X stage* of LEON3. The *X stage* supports precise trap handling, and transitions between processor modes. These extensions enable two crucial elements of our fault handling method. First, secure traps of FAME are immediately handled when the alarm is

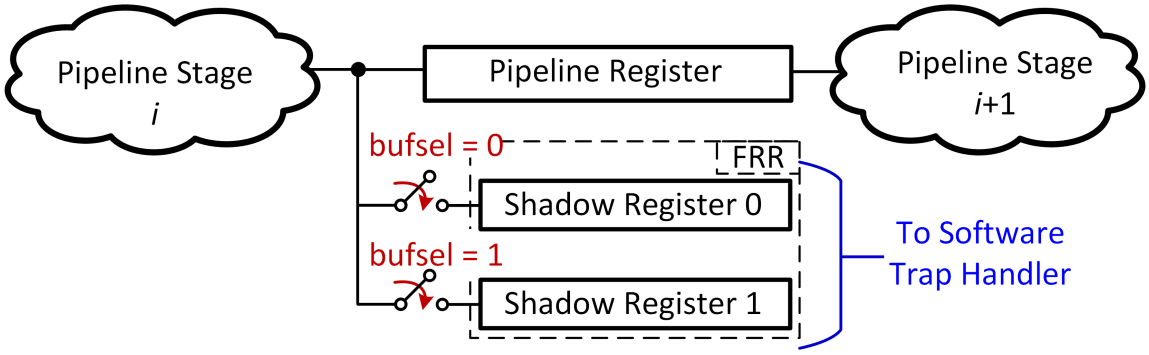


Figure 7.8: Ping-Pong buffering for FRRs: Only one shadow register is updated at a time. Content of FRR is frozen in case of an alarm.

asserted. Second, FAME saves the fault recovery information into FRR and provides this information to the software trap handler for correct execution. Next, we explain our selection and security strategies for the content of FRR.

Fault Response Registers (FRR)

FRR keep the part of the processor state that is updated in C_b, just before the fault injection in C_f. The software trap handler can restore this processor state back and resume the execution of the application.

Fig. 7.8 shows the principle of our FRR implementation. FRR keep the previous value of the original pipeline register in one of its shadow registers; while keeping the new value in the other shadow register. Every clock cycle, only one of the shadow registers is updated. The shadow register to be updated is selected by a 1-bit signal *bufsel*. If `Shadow Register 0` is updated during the *before-fault cycle* C_b, `Shadow Register 1` is updated during C_f. Therefore, it is guaranteed that the fault occurring in C_f cannot contaminate the correct value within both shadow registers. When the alarm is asserted (in C_a), the update of the shadow registers are frozen

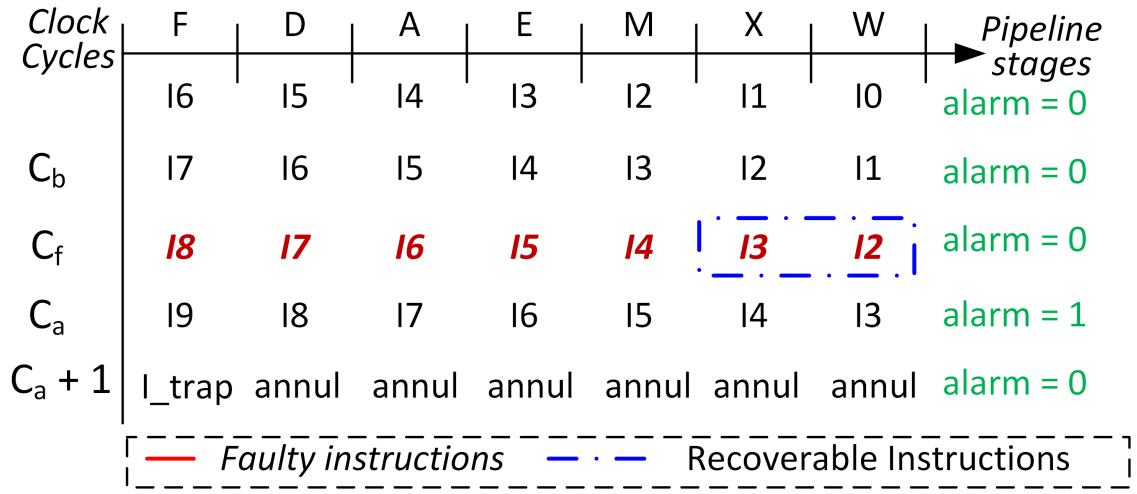


Figure 7.9: Fault effect on the pipeline. Fault is injected during C_f. The alarm is raised during C_a. The first trap handler instruction is fetched after C_a.

until the trap handler is successfully completed. This prevents the correct FRR content from being overwritten after C_f.

We determine the content of FRR by analyzing the effect of the fault injection on the execution of the pipeline. Fig. 7.9 shows the effect of a fault on the LEON3 pipeline. In Fig. 7.9, clock cycles run from the top, and pipeline stages run from left to right. In C_f, up to seven instructions, *I2* – *I7*, will potentially be faulty. During C_f, two instructions could commit their results to the software-visible state of the processor. First, instruction *I4* could write a faulty value to the data cache. Second, instruction *I2* could update the Processor Status Register (PSR) and the register file with a faulty value. Both of these updates need to be intercepted and corrected by the software trap handler. Then the execution can be resumed from the next valid instruction (*I3* in Fig. 7.9). Therefore, FRR keep (a) the register-file write address, write data, and write enable fields of the write-back stage registers; (b) the flags field of PSR; and (c) the address of the instruction being executed in the X stage in C_f.

After control is passed to the software trap handler, it reads the frozen content (a)–(c) of FRR. At the minimum, the trap handler will restore the correct processor state (using (a) and (b)), and resume execution (using (c)).

SPARC instruction set architecture allows up to 32 implementation-dependent registers, called *Ancillary State Registers (ASR)*. Therefore, we integrate FRR into the baseline LEON3 core as ASRs. During the transition from nominal mode to safe mode, the processor hardware writes the program counter to the ancillary state registers `%asr28-29`. We pack the remaining bits of FRRs into two pairs of ancillary state registers `%asr20-21` and `%asr22-23`. The frozen value of *bufsel* is also written into `%asr20-21`. Then the trap handler can know which shadow register contains the correct value.

In SPARC architecture, *Read Ancillary State Registers (RDASR)* is used to access the ASRs. Thus, we use `RDASR` instruction of SPARC as *Read FRR* instruction of FAME. Similarly, we use *Write Ancillary State Registers (WRASR)* instruction of SPARC as *Write FRR* of FAME.

Software Trap Handler

The trap handler is given control of the processor once a fault alert triggers. The secure trap handler can provide different options for handling the fault. One option can be to use the contents of FRR and resume the program under attack from the point of fault injection. In this scenario, we follow the flowchart in Fig. 7.10. First, the trap handler reads `%asr20-21` using `RDASR`. Then it checks the *bufsel* bit in `%asr20-21` to know which shadow registers of FRRs contains the correct value. If *bufsel* is a one, then the content of Shadow Register 0 is valid. If *bufsel* is not set,

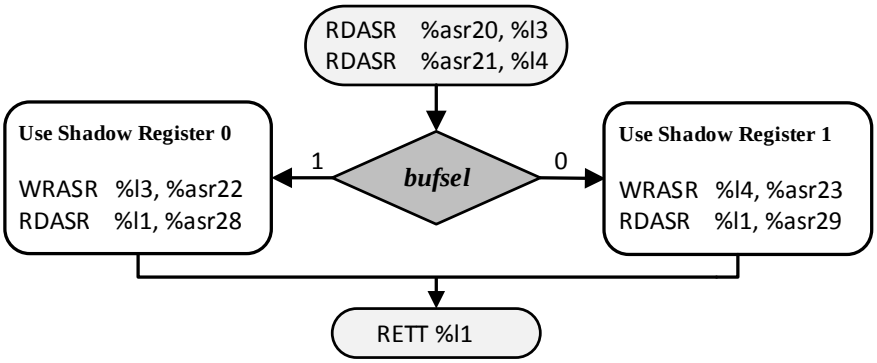


Figure 7.10: Trap Handler Flowchart for Invoking Resume Security Policy

then Shadow Register 1 must be used. Next, the valid FRR is bit-masked to get the register index from it. This register index is the last register that was written to the register file and could have been affected by the fault in C_f. This is written back to the register file through a `WRASR` instruction of LEON3. Our control hardware will use this register index to restore the respective register to its last correct value. Finally, the trap handler restores the PC and returns to the nominal mode for resuming the program.

7.4.4 Fault Analysis Features of FAME SoC

The fault-attack analysis features of the chip allow controlling the fault injection and an in-depth analysis of its effects.

Coprocessors

Several coprocessors are integrated into the FAME SoC to investigate different aspects of fault attacks in the SoC context. One aspect is a detailed cost and security analysis of different sensor types. This would enable secure microprocessor designers to determine best sensor configuration for a given attack vector. For this purpose, we designed a sensor coprocessor consisting of 32 timing sensors and 160 EM sensors distributed over the entire chip. The previous section already explains the security-related part of this coprocessor. The coprocessor has a user interface to configure the sensitivity of the sensors, to enable/disable the sensors, and to observe the status of sensors. One can use this interface for experimenting various sensor configurations under different fault injection experiments.

Another aspect is analyzing the fault effects on the dedicated cryptographic accelerators and their interaction with the FAME core. For this purpose, FAME contains two coprocessors implementing Advanced Encryption Standard (AES-128) algorithm. One implementation, the AES coprocessor, is protected by the distributed sensors of the sensor coprocessor. on the other hand, AES+ coprocessor employs in-situ EM detectors. This setting enables one to investigate the efficiency of different detection strategies. In addition to the AES and AES+ coprocessors, we also integrated an LR-Keymill coprocessor, which is a side-channel-resilient keystream generator. This coprocessor can be used to examine fault effects on a side-channel-protected design and to investigate potential combinational attacks.

The hardware/software interface of the coprocessors is implemented using memory-mapped registers on the APB peripheral bus. We allocate a 4Kbyte address space for each coprocessor, which can be organized as 1024 32-bit memory-mapped registers.

The coprocessor can use the allocated registers for input data, output data, status data, and commands to control the coprocessor's operation. The following sections describe the programming model for these coprocessors.

Debug Support Interface

Our prototype employs an on-chip debug support unit to put the processor into a debug mode and control it in this mode. It also contains a 1-Kbyte instruction trace buffer that stores the executed instructions. The debugging interface connects the microprocessor to a computer for real-time, in-system programming and debugging through Universal Asynchronous Receiver/Transmitter (UART). The debugging controller interfaces with a host computer, which runs a debug monitor used to debug LEON3 based SOC designs [126]. The debug software along with the debug interface is used to load the compiled-C code and program data into the program memory. The debug support interface provides access to all software-visible states of the processor including instruction trace buffer, the register file, cache memories, and the main memory.

Configurable Trigger Unit

In a typical fault attack experiment, fault injection device waits for a trigger signal from the target device (FAME in our case) to start fault injection. This allows control over the timing of injected faults. We extend the baseline LEON3 architecture with Configurable Trigger Unit to generate non-invasive, cycle-accurate trigger signals for fault injection. The trigger unit has 4 independent trigger output that are generated based on the FAME's program counter value `fetchPC` as it is shown in Fig-

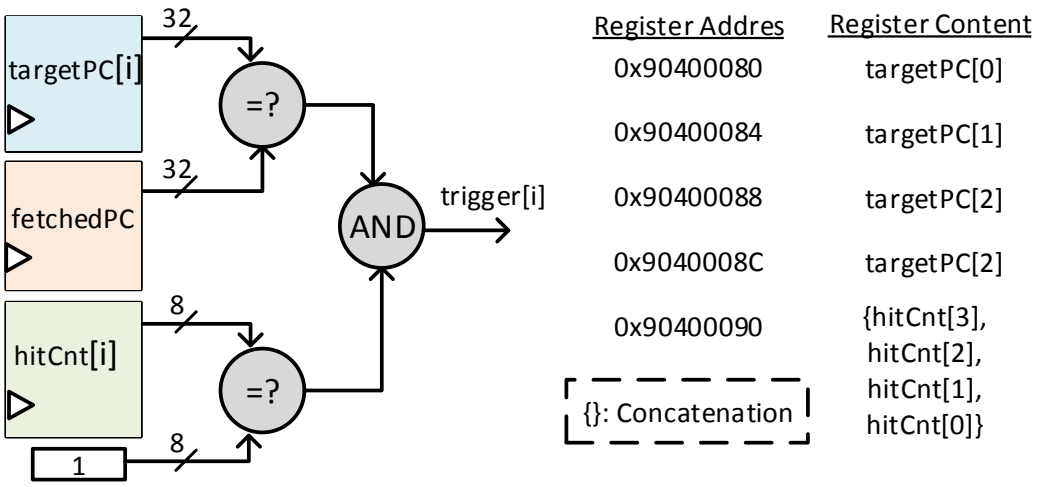


Figure 7.11: Configurable Trigger Unit

ure 7.11. Each trigger output `trigger[i]` is associated with a *target program counter* (`targetPC[i]`) and a *hit counter* (`hitCnt[i]`). At each clock cycle, `targetPC[i]` is compared with the `fetchPC`. The `hitCnt[i]` is an 8-bit counter that is used to specify how many times an instruction from the `targetPC[i]` should be fetched before raising the trigger output `trigger[i]`. We implement each `targetPC[i]` value as a separate 32-bit memory-mapped register and pack all `hitCnt[i]` value into a single 32-bit memory-mapped register (Figure 7.11). All of the memory-mapped registers can be programmed at run-time through the debug interface. We highlight that this method does not require any modification of the software-under-test.

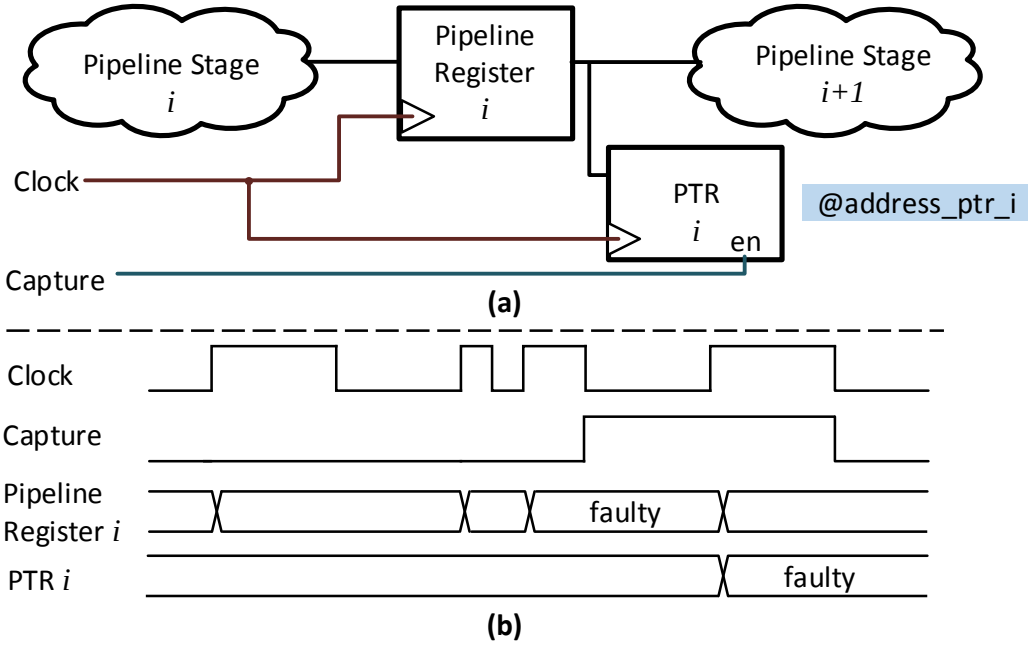


Figure 7.12: Pipeline Trace Registers (PTRs): (a) Block diagram. (b) Timing diagram.

Pipeline Trace Registers (PTRs)

For detailed analysis of the fault injection on a microprocessor, we extend the software-visible state of baseline LEON3 core with Pipeline Trace Registers (PTRs), memory-mapped and read-only registers. They are able to store the values of originally-software-invisible signals in FAME's 7-stage pipeline. As our FAME prototype is a 32-bit processor, we combined individual pipeline signals into 32-bit groups and mapped each group to a unique memory address within the address space of debug support unit. As a result, we implement PTRs as 26 32-bit registers mapped into the address range of `0x90400098 - 0x904000FC`. Those registers can be accessed from the user's C code as well as Tcl scripts executed in the debug monitor software.

Figure 7.12a shows a block diagram, in which a pipeline register i and its corresponding $PTR\ i$ are demonstrated. The memory address for $PTR\ i$ is $@address\_ptr\_i$. Figure 7.12b shows a timing diagram for the operation of the PTR. To save the faulty pipeline signals into PTR, an externally generated *capture* signal is used. When the *capture* signal is asserted, the contents of pipeline registers are copied into the PTRs. When the *capture* goes low, it will freeze content of the PTR register and stored data can be read out by the user software or the debug interface. Using PTRs, we are able to observe all pipeline signals at a time. To integrate PTRs into the FAME prototype, we used the same structure as the base LEON3 uses for its special-purpose registers. Therefore, the area overhead of the PTRs is minimal.

A user can control the *capture* signal in two ways. In the first option, a user drives the allocated external pin of the chip. For this purpose, the user can use an external signal source or GPIO pins of the chip. As a second option, we allocated one of the configurable triggers to generate an on-chip *capture* signal. This option allows a cycle-accurate control of the *capture* signal from the debug monitor software.

External Alarm Pin

FAME chip has an external alarm pin to trigger the secure trap mechanism without physical fault injection. This pin can be used for on-chip fault injection simulation similar to the approach used by Berthier et al. [142]. A user can trigger the secure trap mechanism using the external alarm pin, inject a fault into the processor's state inside secure trap handler, and then continue the execution to observe the effects of the injected fault. This allows evaluating the security of a fault handling strategy under different fault models without using a physical fault injection equipment.

The next chapter provides implementation results for the developed FAME prototype.

Chapter 8

Experimental Evaluation of FAME

This chapter presents the experimental evaluation results for the developed FAME prototype described in Chapter 7. The next section presents the designed fault injection and analysis environment. Section 8.2 shows hardware cost of FAME SoC prototype and FAME core. Section 8.3 provides driver applications to demonstrate how a software designer can use FAME to design fault attack countermeasures and their overheads. Finally, Section 8.4 demonstrates security evaluation results.

8.1 Experimental Setup

The experimental setup consists of a SAKURA-G board and a FInalyzer board, a custom-made evaluation board for fault injection and analysis on the FAME prototype. The FInalyzer board is designed and implemented by Abhishek Bendre, a student member of the FAME development team. It is developed as an extension to the cryptographic standard SAKURA-G Board[123] and attaches to it using the standard expansion headers (Figure 8.1a). This makes FInalyzer fully-compatible with our fault injector, which is described in Chapter 3.

The complete fault injection and observation environment is illustrated in Figure 8.1b. It includes a control PC, fault-attack resistant chip, fault-injector module and an

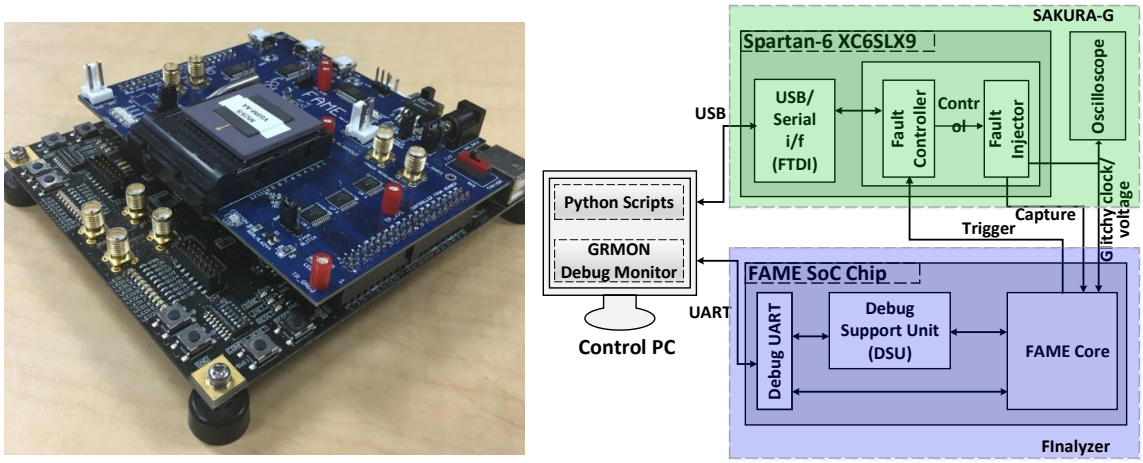


Figure 8.1: (a) Fault-resistant processor die package on FInalyzer board(blue), It is connected to SAKURA-G board(green) (b) Block diagram of fault-injection and analysis setup.

oscilloscope. The fault-injector module is implemented in Spartan-6 FPGA which resides on the SAKURA-G Board. The Fault-Injector FPGA controls the clock and voltage to the fault-resistant processor and can precisely inject faults using the trigger-based glitch mechanism. The fault-resistant chip mounts on a 108-pin Pin Grid Array (PGA) package, which is socketed on a custom testing board as shown in Fig 8.1a.

The PC governs the fault-injection process by configuring the fault-injector module over USB link. Fault-attack injection parameters are dynamically controlled using Python scripts through partial reconfiguration of the injection module. The PC also connects to the fault-resistant processor using the debugging interface loading the compiled software. The processor can generate a trigger signal to inject controlled faults, and it allows the fault-injector module to inject a fault with specified intensity parameters.

Table 8.1: Area and Power Consumption Results for FAMEv2

Component	**Cell Area** (sq. μ)	**Cell Area** (GE NAND2X1)	**Power** (mW)
FAMEv2 SoC	12,332,181	1,235,789	190.899
SRAM macro's	9,677,474	969,764	34.917
Sensor Coprocessor (CP1)	199,514	19,993	11.056
AES Coprocessor (CP2)	627,196	62,850	45.627
AES with in-situ sensors (CP4)	739,958	74,150	57.659
LR-Keymill Coprocessor (CP3)	121,893	12,215	3.904
FAMEv2 Core	1,864,244	186,813	52.463
AHB Bus Controller	13,841	1,387	0.541
APB Bus Controller	12,368	1,239	0.433
User UART	20,397	2,044	1.011
Debug UARTs	20,743	2,079	1.734
Debug Support Unit	8,971	899	0.741
Timer	54,709	5,482	2.302
SPI Interface	17,091	1,713	0.887

8.2 Hardware Performance Results

This section presents the hardware performance results of the FAME prototype including cell-area, power consumption, and operating frequency.

8.2.1 Performance Results for FAME SoC

FAME SoC chip was fabricated in a TSMC 180nm process on a 5 mm by 5 mm die. The chip was constrained for 80MHz operating frequency during synthesis and back-end design. Table 8.1 lists the active area and power consumption results for the major components of the FAME SoC, which is a 1.2M GE design. The results are collected from the post-synthesis design using Synopsys DC.

Table 8.1 also allow us to evaluate the overhead of the in-situ EM sensors in the AES+ coprocessor (`cp4`) by comparing the results for `cp2` and `cp4`. Although `cp4` replaces all the critical state elements of `cp4` with in-situ EM detectors, the sensors have less than 20% and 30% overhead in area and power consumption, respectively. This overhead includes both the fault-attack-related and fault-analysis-related cost of the in-situ detectors. Next section examines the hardware overhead of FAME extensions on the processor core.

8.2.2 Performance Results for FAME Core

The secure extensions added to the processor core have associated costs. The Fault Detection Unit (FDU), Fault Response Registers (FRR), Fault Control Unit (FCU), FAME instructions for FRR access are the only logic added to the processor core to make it fault-attack resistant. However, the ASIC prototype also contains injection and analysis capabilities to enhance the fault-attack environment. Therefore, we compare the cost of chip with and without the additional units with respect to the base processor. We obtain area and timing overhead for (a) LEON3 implementation without extensions (b) LEON3 implementation with fault-attack resistant security extensions, and (c) LEON3 implementation with fault-attack resistant and analysis extensions. The cost is derived by applying the same design process on different design configurations. Thus, we do ASIC synthesis, verification and then place-and-route the design using Synopsys Suite (Design Compiler, ICC Compiler) at 180 nm TSMC logic process to get the required design metrics.

Table 8.2 presents the resulting area and power consumption performance. Fault analysis extensions incurs a constant area and power consumption overhead of 0.124

Table 8.2: Hardware Overhead of FAME Extensions

Component	**Cell Area** (sq. mm)	**Cell Area** (GE NAND2X1)	**Power** (mW)
LEON3 Core (Baseline)	1.709	171,257	43.149
LEON3 + Fault Analysis	1.833 $\Delta = 0.124$	183,682 $\Delta = 12{,}425$ (%6.7)	49.351 $\Delta = 6.202$ (%14.4)
LEON3 + Fault Resistance (FAMEv2)	1.740 $\Delta = 0.031$	174,363 $\Delta = 3{,}106$ (%1.8)	46.261 $\Delta = 3.112$ (%7.2)

mm^2 (i.e, 12.4K GE) and 6.202 mW, respectively. Thus, the area and power consumption overhead of the fault analysis features on the baseline LEON3 implementation is 6.7% and 14.4%, respectively. For the security extensions, the respective are and power consumption overhead on the baseline LEON3 are 1.8% and 7.2%. Moreover, the total area cost of security and analysis extensions is less than 10% and their power consumption overhead is less than 20%. The maximum operating frequency is same, 80 MHz, in all of the investigated cases.

As the FAME chip was designed both fault-attack-resistance and fault-attack-evaluation in mind, it contains redundant control and observation logic for the sensors in addition to security-related logic of the sensors. Table 8.3 provides cell area and power consumption results of the security-related parts of the timing end EM sensors for a rough estimation of each sensor's cost. As it is seen, the area and power consumption of each sensor is less than 0.2 that of FAME core. The cost of the stand-alone EM sensors of the `cp1` is more than the cost of the in-situ EM sensors of the `cp4`. The reason behind this difference is that the input of the two flip-flops of the in-situ EM sensors are directly connected to the output of the combinational logic of the AES design. However, in the case of `cp1`, an additional inverter and a flip-flop (i.e, toggle

Table 8.3: Hardware Overhead of FAME Sensors

Component	**Cell Area** (sq. μm)	**Cell Area** (GE NAND2X1)	**Power** (μW)
Timing Sensor (CP1, Core)	2,810.808	282	129.530
Stand-alone EM Sensor (CP1)	182.952	19	22.406
In-situ EM Sensor (CP4)	113.098	12	21.900

flip-flop) are connected to the input of the stand-alone EM sensor to toggle its input at every clock cycle.

8.3 Software Performance Results

This section first demonstrates a methodology to develop FAME-protected software programs on three example applications. Then, it presents the software overhead of the FAME protection.

8.3.1 FAME-Protected Software Design

In this section, we describe a few secure trap handlers and compare these with traditional redundancy-based software countermeasures. We consider a fault countermeasure scenario for three different applications: PIN Verification, the Advanced Encryption Standard (AES), and a Pseudo-Random Number Generator (PRNG). For each of these, we will create an application-specific secure trap handler.

Redundancy-based fault countermeasures are based on customization of the application code. In our case, however, the secure trap handler is created *separately* from the application, and it follows a standard development methodology based on the

following steps.

1. *Identify the Undesirable Outcome:* We consider potential security-failure scenarios for the given fault-injection attack vector.

2. *Analyze the Code Sensitivity:* We analyze the application code to identify how the undesirable outcome can be achieved through fault injection. This identifies sensitive points in the code such as security-critical decisions and operations that process and transfer the sensitive data.

3. *Design the Countermeasure:* We write a fault handler to mitigate the undesirable outcome. The fault handler must be effective regardless of the current state of the protected algorithm, because the secure trap can be invoked at any moment.

Case Study I: PIN Verification

A Personal Identification Number (PIN) is an alphanumeric passcode used for authenticated access. Figure 8.2a shows an unprotected PIN verification function. `VerifyPin()` function compares a user PIN `userPIN` to a secret value `devicePIN`. The verification allows up to three mismatches, and this is recorded in the variable `counter`. When `userPIN` and `devicePIN` match, the user accepted, otherwise the `counter` is decremented. We apply the methodology to develop a secure trap handler.

1. *Undesirable Outcome:* There are two undesirable results. First, an adversary can successfully complete `VerifyPin()` with an *invalid* PIN. Second, an adversary can attempt a PIN check without losing a trial from `counter`.

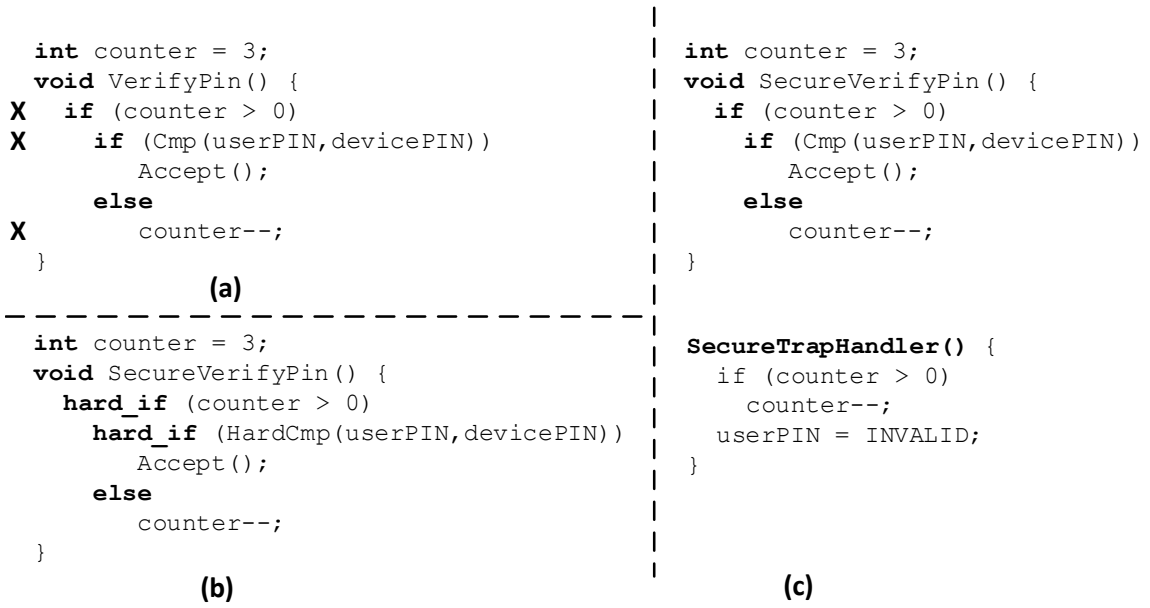


Figure 8.2: PIN Verification (a) Unprotected version. The lines marked with **X** are sensitive points. (b) Fault protection with software redundancy. (c) Fault protection with proposed secure trap technique.

2. *Code Sensitivity:* There are multiple methods to achieve the undesirable outcome. The adversary can target the data-flow or control-flow of the `if` statements or the `Cmp()` function invocation, skip the decrement of the counter value, or invert the result of the `Cmp()` function. All of these can be achieved using glitch injection under the assumed fault model. The sensitive lines of code are marked in Figure 8.2a with a cross (**X**).

3. *Countermeasure Design:* Figure 8.2b shows a protected `VerifyPin()` that uses traditional software redundancy. The `if` statements can be hardened (`hard_if`) by using double checks [166]. The execution of `Cmp()` function can be hardened (`HardCmp()`) by duplicating its instructions, and subsequently checking if the results match [66]. However, these methods bring significant overhead as we

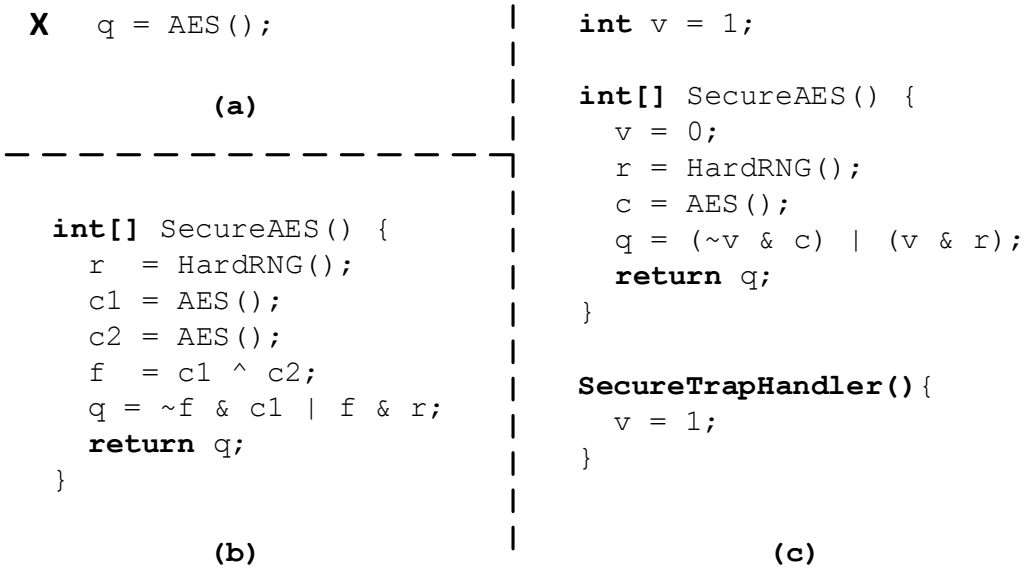


Figure 8.3: (a) Unprotected AES. The lines marked with X are sensitive points. (b) Fault-protected AES using software redundancy. (c) Fault-protected AES using the proposed secure trap technique.

show in Section 5.

On the other hand, the proposed secure trap handler technique leads to Figure 8.2c. Whenever a fault injection is detected, the secure trap handler decrements the `counter` and sets the `userPIN` to an invalid value. We do not need hardened `if` statements in `VerifyPin()` because the fault-resistant processor provides inherent immunity against instruction-skip attacks through the FRR mechanism (Section 2).

Case Study II: Advanced Encryption Standard

The second case study considers protection of AES against fault attacks.

1. *Undesirable Outcome:* The secret key of AES can be extracted using various fault analysis techniques. Standard DFA techniques rely on observation of faulty ciphertext, and therefore the disclosure of faulty ciphertext is an undesirable outcome. However, the detection that any fault occurred is already sufficient for Fault Sensitivity Analysis[77]. Although FSA on software may require a fine-grained instruction-level fault analysis[12], we do not discount this risk.

2. *Code Sensitivity:* Fault injection in AES has been extensively studied, and successful attacks have been shown based on injecting faults into round computations of AES [167], key scheduling of AES [167], or increasing/decreasing the number of iterative rounds [168]. Therefore, the entire AES procedure is sensitive against fault attacks.

3. *Countermeasure Design:* Classic fault countermeasures for AES rely on time-based, information-based, or hybrid redundancy [169]. Figure 8.3b shows a time-based redundancy design. If two redundantly computed ciphertext `C1` and `C2` are different, the `SecureAES()` returns a random value. Otherwise it returns the actual ciphertext. However, this countermeasure is still ineffective against FSA.

   Figure 8.3c shows a solution using the proposed secure trap technique. In case a fault is detected, a flag `v` is set which will cause randomization of the ciphertext. This will also catch FSA-style attacks, because the sensor has a higher fault-sensitivity than the processor hardware. The proposed approach is also more performance-efficient, since we do not need a redundant execution of AES.

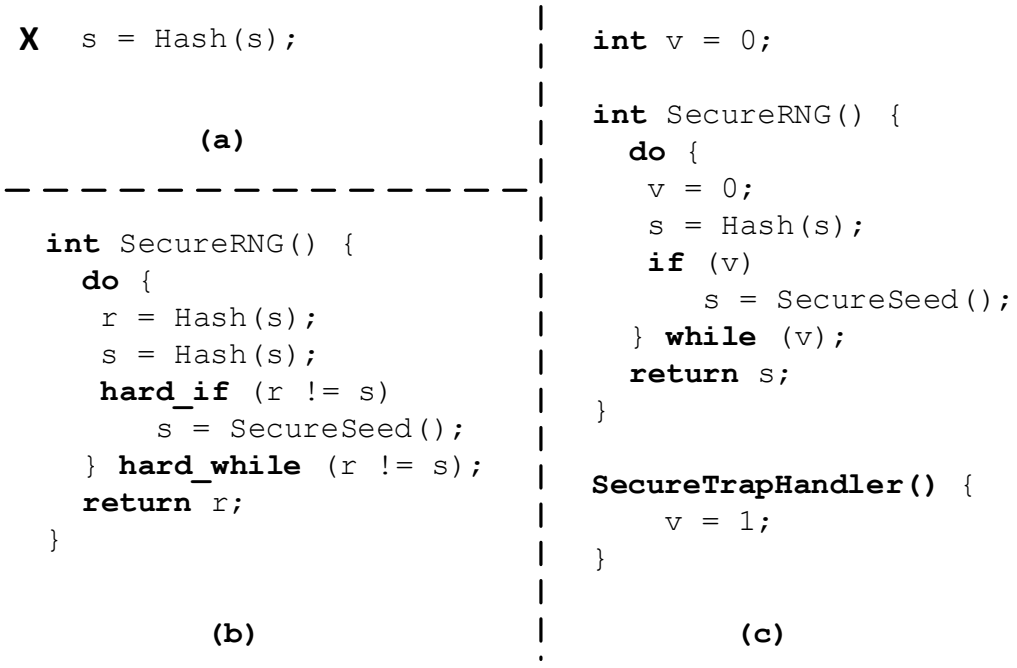


Figure 8.4: PRNG with backtracking resistance: (a) Unprotected case. The lines marked with X are sensitive points. (b) Fault-protected version using software redundancy. (c) Fault-protected version using the proposed secure trap technique.

Case Study III: Pseudo Random Number Generator

Pseudo Random Number Generators (PRNGs) are an essential component in many crypto-systems, for example to ensure freshness in crypto-protocols or to ensure randomness in mask generation for secret sharing. We apply the methodology to develop a secure trap handler for PRNGs. In this example we study a PRNG with backtracking resistance but no prediction resistance [170]. This type of PRNG is used in applications that do not disclose the random output.

1. *Undesirable Outcome:* A fault-attack on a backtracking-resistant PRNG will aim at introducing bias or at controlling the secure state.

2. *Code Sensitivity:* An adversary can inject stuck-at faults to control the internal state of the PRNG. She can also prevent update of the PRNG.

3. *Countermeasure Design:* A general strategy against securing a PRNG is to re-seed it when a fault is detected. The new seed comes from a source of secure entropy (`SecureSeed()`), for example created using a true random number generator and tested for randomness. The design of the source of secure entropy is out of the scope of this paper, and we concentrate on protecting the PRNG.

   In Figure 8.4b, we demonstrate a time-redundancy based countermeasure. In this countermeasure, the PRNG code is executed twice, and the results are compared. If a difference is found, the PRNG is reseeded and the loop is repeated. Similar to the case of PIN Verification, this code requires hardened `if` and `while` statements [166], and the time redundancy will reduce performance.

   Figure 8.4c shows a solution using the proposed secure trap technique. We run the PRNG once while capturing faults in a flag `v`. When a fault is injected, we re-seed the PRNG and force the PRNG output to be recomputed.

The described case studies clearly highlight the differences between the redundancy-based and FAME-based approaches for countermeasure design. In the redundancy-based countermeasures, the software designer employs generic redundancy directly in the application code for fault detection and fault response. In addition, the software designer does not distinguish an application-specific undesired outcome. In contrast, in the FAME-based approach, the software designer partition the fault detection and response into hardware and software. The fault countermeasure is developed separately in the trap handler.The designer utilizes the generic architectural support of FAME to fine-tune the fault countermeasure for the specific security requirements

of the protected application. For instance, the designer uses the software trap handler to enforce the decrement of `counter` in the PIN Verification case, to randomize the cipher output in the AES case, and to reseed the random number generator in the PRNG case. Next, we examine the cost of FAME on the software layer.

8.3.2 Software Overhead of FAME Extensions

This section demonstrates the software overhead of the proposed method for the described case studies. We implemented the pseudo-codes shown in Figures 8.2-8.4, and examined their code size and the execution time.

In the Case Study I (i.e, PIN Verification), we hardened the `if` statements with double checks [166], and we used instruction duplication countermeasure [66] to protect the data and control flow of the pin comparison function `Cmp()`. For the Case Study II, we implemented the `AES()` function with the well-known TBOX method, which combines multiple steps of an AES round (comprising SBOX-lookup, Shiftrows and Mixcolumns) into a single lookup. We used SHA-1 hash algorithm to estimate the software overhead for the Case Study III (i.e, PRNG).

The code of secure trap handler consists of three parts. In the *prelude part*, the trap handler restores the critical fault-free state back by using the FRR. It also gets the return address from FRR. Then, in the *body part*, the trap handler applies the user-defined fault response. In the *postlude part*, the trap handler jumps to the return address and continues the execution from there. As the prelude and postlude parts are the same for all case studies, their code-size overhead is constant: The code size of prelude and postlude is 132 *bytes* (i.e, 32 instructions). The code size of the trap handler body and its performance overhead depend on the actual fault response

Table 8.4: Software Code-Size Overhead of FAME (Byte)

Case Study	Unprotected	Protected w/ Redundancy	Protected w/ FAME
Pin Verification	300	+ 702 (234%)	+ 152 (52%)
AES	5,344	+ 344 (5.9%)	+ 148 (2.7%)
PRNG	1,064	+ 560 (52.6%)	+ 292 (27.4%)

Table 8.5: Software Execution-Time Overhead of FAME (Cycle Count)

Case Study	Unprotected	Protected w/ Redundancy	Protected w/ FAME
Pin Verification	1,013	+ 1,628 (160%)	+ 0
AES	7,130	+ 6,080 (85%)	+ 728 (3.2%)
PRNG	1,7213	+ 18,226 (106%)	+ 925 (5.4%)

policy.

Table 8.4 shows the overhead of traditional software redundancy and the proposed method on the code size of the protected applications. For the case studies, the proposed method has 2 to 4.5 times less code-size overhead in comparison to the traditional method. In addition, the code-size overhead of the secure trap handler is *independent of the protected application's code size*. However, the code size of the traditional approaches proportionally increases with the size of the application because they apply redundancy to the application's code itself. In Table 8.4, this is more prominent for the PIN Verification case because it duplicates all instructions of the comparison function `Cmp()`. However, in the other case studies, the selected traditional approaches only duplicates the control statements such as branches and loops, and executes the application twice.

Table 8.5 lists the performance overhead of the examined countermeasures on the execution time of the unprotected application. In all of the case studies, the per-

formance overhead of the traditional countermeasures (85%–160%) is significantly higher than the performance overhead of our method (0%–5.4%). The main reason behind this observation is that a software designer has to write the fault detection and response in the application code for the traditional approach. However, in our method, the software designer takes advantage of the hardware fault detection and writes the fault response in the secure trap handler. This significantly lowers the performance overhead. Our method does not require to modify the application code (i.e, `VerifyPin()`, `AES()`, and `Hash()`). On the other hand, there is a minor overhead because of the introduction of a wrapper around the basic protected function.

8.4 Security Evaluation of FAME

In this section, security-related experimental results are presented. First, the fault detection sensitivity of the timing sensor is characterized under different operating conditions. Then a fault attack is applied on both FAME-protected and redundancy-protected PIN Verification applications.

8.4.1 Fault Detection Sensitivity

This section provides characterization results for the timing sensor of the FAME prototype. The characterization results for the EM sensor can be found in Master Thesis of Chinmay Deshpande [162], a member of FAME development team.

The sensitivity of the Fault Detection Unit is evaluated by injecting faults with different parameters into the fault-resistant ASIC and by checking its response. Specifically, we inject a clock-glitch to violate setup-time and then examine the response

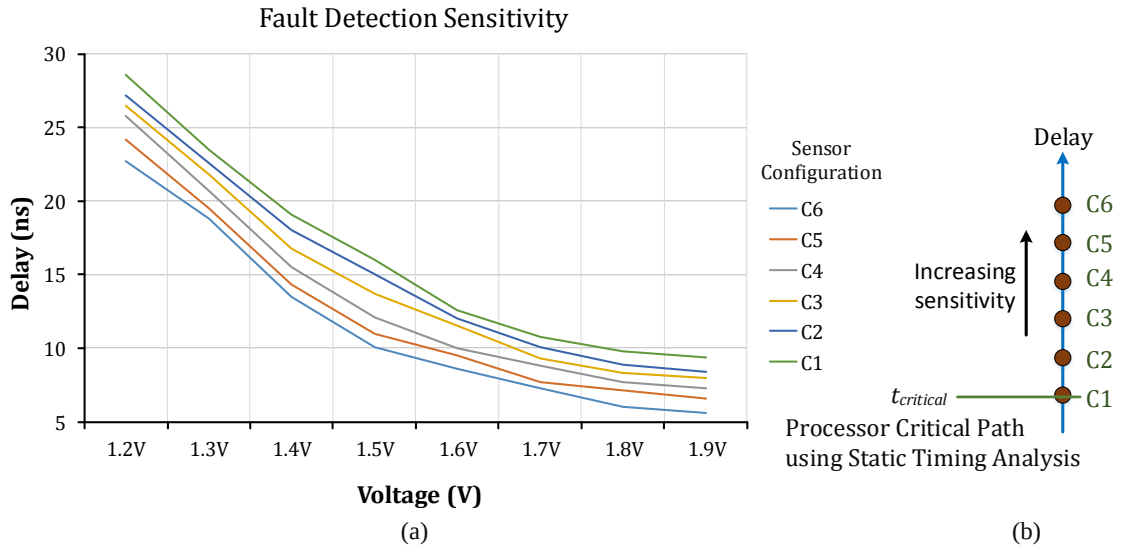


Figure 8.5: (a) Fault Detection Sensitivity of the sensor for different configurations ($C1 - C6$) (b) Static Timing Analysis results of the processor core and sensor for different configurations

of the detection unit. The clock glitch width is then gradually changed and the response is then recorded for different sensor configuration settings. This procedure is carried out at different power levels by changing supply voltage from $1.2V$ (the lowest correct operating voltage) to $1.98V$ (maximum operating voltage) in increments of $0.1V$. The recommended normal operating voltage for this specific standard cell library is $1.8V$.

Fig 8.5a shows the fault detection sensitivity of a timing sensor measured over a single chip. The sensor has about sixty different sensitivity settings (six of them are shown), and they help to minimize effects of intra-die delay variation between the CPU core and the timing sensor. Fig 8.5b shows the design-time static timing delay for the CPU and the sensor settings. The sensor setting should to be chosen such that it has a higher sensitivity than the critical delay of the CPU core. But

in order to minimize the number of false positives (ie. triggering the sensor without an actual fault in the CPU), the sensor sensitivity should also match the critical delay of the CPU core as closely as possible. The setting of the sensor sensitivity is a *security setting* that should be treated with the same care as an embedded secret key. While the current prototype supports sensor setting through the debug interface, we envisage that a production version of this chip would use secure fuses.

8.4.2 Clock Glitching on PIN Verification

To demonstrate the efficiency of our approach, we have conducted a clock glitching experiment for the pin verification case described in Section 4. `VerifyPin()` function compares a user PIN `userPIN` to a secret value `devicePIN`. The verification allows up to three mismatches, and this is recorded in the variable `ptc`. For this example, `devicePIN` is `12824`. When `userPIN` and `devicePIN` match, the user is accepted (i.e, `result = 1`), otherwise the `counter` is decremented.

Figure 8.6a–c shows three versions of the `VerifyPin()` and corresponding clock glitching experiments: (a) No fault protection, (b) Fault protection with software redundancy, and (c) Fault protection with FAME. In this experiments, the attack objective is to successfully complete the `VerifyPin()` function with an invalid `userPIN`. We have used our experimental setup to inject faults in to the attack windows shown with red rectangles in Figure 8.6.

We start with the unprotected `VerifyPin()`. Figure 8.6a shows the clock glitch that enables us to break the `VerifyPin()` with 100% success rate. As it is seen from the example software output for two different `userPIN` values, `ptc` has not been decremented and `result` is 1 although the values are wrong.

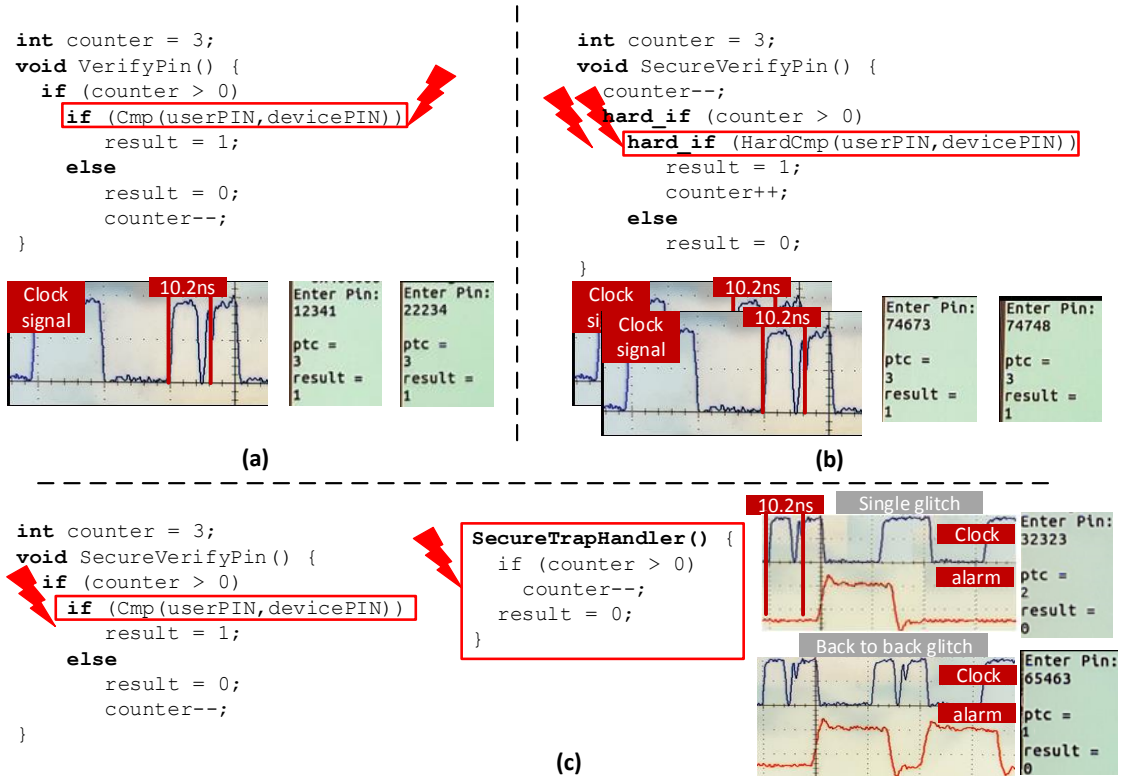


Figure 8.6: Clock glitching on `VerifyPin()`: (a) Unprotected, (b) Protected with software redundancy, (c) Protected with FAME. `devicePIN` is `12824`. Protections in (a) and (b) fail: `ptc` has not been decremented and `result` is 1 although the `userPIN` values are wrong. FAME is secure against single and multiple glitches.

Figure 8.6b shows a `VerifyPin()` function protected with software redundancy: This implementation runs the comparison function and check its result twice. It also decrements the `ptc` by default and increments it if `userPIN` and `devicePIN` match. We have successfully bypassed this protection by repeating the previous glitch after each execution of the comparison function. Figure 8.6b shows the results for two successful cases.

Finally, we have attacked the FAME-protected version of `VerifyPin()` (Figure 8.6c).

Whenever a fault injection is detected, the secure trap handler decrements the `ptc` and sets the `result` to 0. We do not need hardened `if` statements in `VerifyPin()` because the FAME processor provides inherent immunity against instruction-skip attacks through its hardware mechanisms explained in Section 3. When we apply the same single glitch, FAME protection makes `result 0` and `ptc 2` as shown in Figure 8.6c. We have also applied two back to back glitches to FAME to demonstrate that it works under multiple glitch injections. In this case, FAME protection makes `result 0` and `ptc 1` as shown in Figure 8.6c.

Chapter 9

Conclusions

This research has investigated fault attacks on embedded software, a serious hardware-oriented threat to the security of embedded systems, from several aspects:

- **Understanding Fault Attacks on Embedded Software:** Chapters 2 and 4 explain the steps and requirements of fault attacks on embedded software. By analyzing a 7-stage RISC pipeline, we have shown that the fault manifestation and propagation in software security systems are quite different than hardware secure systems. In modern software secure systems, faults manifest in the hardware layers but their effects are observed in the software layer. Modern microprocessors execute each instruction as a sequence of instruction steps (i.e, fetch-decode-execute cycle), and generally, multiple instructions are executed in parallel (e.g, pipelining). During our analysis we have made the following observations:

    1. A fault injection attempt may potentially affect multiple instructions that are being executed during the fault injection.
    2. The effects of fault injection depend on the affected instruction(s) as well as affected instruction step(s). For instance, a fault injection into execution step of a memory-load instruction would alter the memory address, while a fault injection into execution step of an addition instruction would

affect the computed result. Similarly, affecting instruction-fetch step of a memory-load instruction may turn this instruction into another one, while affecting the execute step of the same instruction alters the address calculation.

3. Microarchitectural and pipelining events (e.g, cache misses, stalls because of data-dependency, branch interlock, etc.) affect the fault sensitivity.

After its manifestation, the effect of a fault is propagated to the output through subsequent instructions that use this fault. Therefore, mounting efficient fault attacks on software systems and mitigating them require the knowledge of different abstraction layers.

- **Experimenting Fault Attacks on Embedded Software:** As the fault manifestation and propagation are complex, we need an advanced fault injection and analysis setup for the experimental part of the research. Chapter 3 describes our custom-made fault injection and analysis setup, which enables us to inject setup-time violation faults through precise clock glitch injections. Our setup has also extensive fault observation capabilities that allow us to observe architectural (software-visible) and micro-architectural (software-invisible) effects of the fault injection. Using this setup, we are able to carry on fault attack experiments on both hardware and secure software systems implemented either on an FPGA or an ASIC chip.

  We have also integrated the fault injection and observation capabilities into the chip prototype of our fault-attack-resistant processor FAME. This integration turns our chip prototype into a complete on-chip fault attack laboratory, which is quite useful for the long-term fault attack research.

- **Modeling Fault Attacks on Embedded Software:** As the existing fault models do not use the knowledge of multiple abstraction layers, they are insufficient to capture fault effects on software and to guide the fault injection process to obtain desired fault effects. In Chapter 4, we have introduced instruction fault sensitivity model for a microprocessor to overcome the limitations of existing fault models.

  The proposed fault model can explain the fault effects experienced by a software program at the instruction-set-architecture level, and can guide the fault injection process to induce the desired fault effects in the program. This model allows us to identify the potential fault effects on every (*instruction*, *pipeline stage*) pair. It also enables us to tune the fault injection parameters to induce the desired effects in each (*instruction*, *pipeline stage*) pair.

  We have experimentally demonstrated the model on an FPGA implementation of the LEON3 processor. We have shown that the instruction fault sensitivity model reduces the number of required fault injections and enables new fault attacks on the software.

- **Designing Novel Fault Attacks on Embedded Software:** Relying on the instruction fault sensitivity model, we have also built a systematic fault attack methodology, so-called Microarchitecture Aware Fault Injection Attack (MAFIA). MAFIA enables an adversary to launch an efficient fault attack on an embedded software by exploiting different layers of abstraction. The adversary starts with algorithm-level analysis to determine application-specific fault injection and analysis objectives. Then the adversary studies the software implementation of the algorithm in the instruction level, and finds the candidate (*instruction*, *pipeline stage*) pairs for fault injection. Finally, the adversary

examines the execution of the instructions on the pipeline to determine clock cycles to inject faults as well as the fault injection parameters to create the desired effects.

We showed the efficiency of MAFIA with two case studies. First, we developed a DFIA attack on an unprotected AES software program, and showed that the use of the proposed methodology reduced the number of fault injections an order of magnitude in comparison to the traditional attack methodology. Second, we studied instruction-level software countermeasures. We identified their vulnerabilities using our method, and demonstrated that all of the studied countermeasures can be broken with single fault injections with low-cost injection setups such as clock glitching. We also experimentally demonstrated the attacks on an FPGA implementation of LEON3 processor. Finally, we conclude that efficient countermeasures to protect embedded software must consider multiple abstraction layers.

- **Simulating Fault Attacks on Embedded Software:** To enable software designers to evaluate their designs against fault attacks, we propose a fault attack simulator MESS (Microarchitectural Embedded System Simulator). MESS is a cycle-accurate simulator that enables a user to model main microarchitectural components of a processor, to model instruction set architecture of the processor, and to run the target software on the modeled processor. A user can also describe a complete fault attack experiments including fault timing, fault location, fault type, attack objectives, and observation points. Currently, it supports ARM and x86 instruction set architectures.

  We have demonstrated the features of MESS on a C library function `memcmp()` that can be employed for security verification. As MESS enables us to reflect

attacker's view of execution, it has allowed us to try different real-world fault attack scenarios.

- **Mitigating Fault Attacks on Embedded Software:** To mitigate fault attacks, we have designed a fault-attack-resistant microprocessor called FAME (Fault-attack Aware Microprocessor Extensions). FAME combines fault detection in hardware and fault response in software. This allows low-cost, performance-efficient, and flexible integration of hardware and software techniques to mitigate fault attack risk. FAME is a generic solution, applicable to existing embedded processors.

  FAME is low-cost as it uses redundancy to protect only a small subset of the processor state (i.e, FRR) and a small portion of the embedded software (i.e, trap handler). FAME extensions do not bring any timing overhead on the processor hardware. On the software side, FAME affects the performance only if a fault injection is detected. FAME enables a flexible and application-specific trap handler, which can be adjusted for the security needs of the application.

  We have created a chip prototype of FAME to explore this concept and to verify our claims. The prototype demonstrates that FAME enables application-specific fault countermeasures with a much lower performance overhead, and with a very low additional hardware cost. As a byproduct of this research, we have created a RISC environment that enables the detailed study of the fault effects occurring in an embedded processor. This may lead to better insight into fault modeling for embedded processors.

In conclusion, our research findings indicate that efficient modeling, design, mitigation, and simulation of fault attacks on embedded software require considering

multiple abstraction layers together. The outcome of this dissertation has been published as 2 journal papers [171, 172], 8 peer-reviewed conference papers [11, 12, 75, 107, 129, 173, 174, 175], and a book chapter [176]. We have also filed a patent application [177] part of this research has been recognized as the best paper in session by the SRC TECHCON community in 2015. Two publications of this research have been awarded the best poster in CESCA Day 2016 and 2017. The author of this dissertation has been recognized as the outstanding student in CESCA Day 2017. The chip demonstrator of FAME has been awarded Best Hardware Demo by IEEE International Symposium on Hardware Oriented Security and Trust (HOST) in 2017. This research has also been presented in PhD Forum at Design Automation Conference (DAC) 2017.

Bibliography

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 48–62.

[2] F. Piessens and I. Verbauwhede, "Software security: Vulnerabilities and countermeasures for two attacker models," in *Design, Automation & Test in Europe Conference & Exhibition, (DATE 2016)*, 2016, pp. 990–999.

[3] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[4] M. Joye and M. Tunstall, *Fault Analysis in Cryptography*. Springer, 2012.

[5] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1997, pp. 37–51.

[6] R. Piscitelli, S. Bhasin, and F. Regazzoni, "Fault attacks, injection techniques and tools for simulation," in *Proc. of DTIS'15*, 2015, pp. 1–6.

[7] A. Barenghi, G. Bertoni, E. Parrinello, and G. Pelosi, "Low Voltage Fault Attacks on the RSA Cryptosystem," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC),*. IEEE, 2009, pp. 23–31.

[8] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[9] S. Ordas, L. Guillaume-Sage, K. Tobich, J.-M. Dutertre, and P. Maurine, "Evidence of a larger em-induced fault model," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2014, pp. 245–259.

[10] J. G. Van Woudenberg, M. F. Witteman, and F. Menarini, "Practical optical fault injection on secure microcontrollers," in *Proc. of FDTC'11*. IEEE, 2011, pp. 91–99.

[11] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont, "Software fault resistance is futile: Effective single-glitch attacks," in *Proc. of FDTC'16*, 2016, pp. 47–58.

[12] B. Yuce, N. F. Ghalaty, and P. Schaumont, "Improving fault attacks on embedded software using risc pipeline characterization," in *Proc. of FDTC'15*, 2015, pp. 97–108.

[13] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller," in *Proc. of FDTC'13*, 2013, pp. 77–88.

[14] T. Korak and M. Hoefler, "On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms," in *Proc. of FDTC'14*, 2014, pp. 8–17.

[15] D. Karaklajic, J. Fan, and I. Verbauwhede, "A Systematic M Safe-error Detection in Hardware Implementations of Cryptographic Algorithms," in *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*, June 2012, pp. 96–101.

[16] C. OFlynn and Z. D. Chen, "ChipWhisperer: An Open-source Platform for Hardware Embedded Security Research," in *Constructive Side-Channel Analysis and Secure Design*. Springer, 2014, pp. 243–260.

[17] N. Timmers and C. Mune, "Escalating privileges in linux using voltage fault injection," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2017)*, 2017, pp. 25–35. [Online]. Available: http://dx.doi.org/10.1109/FDTC.2016.18

[18] N. Timmers, A. Spruyt, and M. Witteman, "Controlling PC on ARM using fault injection," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2016)*, 2016, pp. 25–35. [Online]. Available: http://dx.doi.org/10.1109/FDTC.2016.18

[19] R. Pareja, N. Wiersma, and M. Witteman, "Safety not security. on the resilence of asil-d certified microcontrollers against fault injection attacks," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2017)*, 2017, pp. 25–35. [Online]. Available: http://dx.doi.org/10.1109/FDTC.2016.18

[20] L. Cojocar, K. Papagiannopoulos, and N. Timmers3, "Instruction Duplication: Leaky and Not Too Fault-Tolerant!" Cryptology ePrint Archive, Report 2017/1082, 2017, http://eprint.iacr.org/.

[21] C. OFlynn, "A framework for embedded hardware security analysis," Ph.D. dissertation, Dalhousie University, 2017.

[22] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the perils of security-oblivious energy management," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association,

2017, pp. 1057–1074. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang

[23] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 300–321.

[24] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," *arXiv preprint arXiv:1710.00551*, 2017.

[25] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1675–1689.

[26] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation." in *USENIX Security Symposium*, 2016, pp. 19–35.

[27] B. Giller, "Implementing Practical Electrical Glitching Attacks," https://www.blackhat.com/docs/eu-15/materials/eu-15-Giller-Implementing-Electrical-Glitching-Attacks.pdf, [Online; accessed 14-Nov-2017].

[28] M. E. Scott, "Glitchy Descriptor Firmware Grab," https://www.youtube.com/watch?v=TeCQatNcF20, [Online; accessed 14-Nov-2017].

[29] J. Balasch, B. Gierlichs, and I. Verbauwhede, "An in-depth and black-box characterization of the effects of clock glitches on 8-bit MCUs," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2011)*, 2011, pp. 105–114. [Online]. Available: http://dx.doi.org/10.1109/FDTC.2011.9

[30] A. Barenghi, G. M. Bertoni, L. Breveglieri, M. Pelliccioli, and G. Pelosi, "Injection Technologies for Fault Attacks on Microprocessors," in *Fault Analysis in Cryptography*, ser. Information Security and Cryptography, M. Joye and M. Tunstall, Eds. Springer Berlin Heidelberg, 2012, pp. 275–293.

[31] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller," *CoRR*, vol. abs/1402.6421, 2014. [Online]. Available: http://arxiv.org/abs/1402.6421

[32] J. Quisquater and D. Samyde, "Eddy Current for Magnetic Analysis with Active Sensor," in *Esmart*, 2002.

[33] L. Riviere, Z. Najm, P. Rauzy, J.-L. Danger, J. Bringer, and L. Sauvage, "High precision fault injections on the instruction cache of armv7-m architectures," in *Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 62–67.

[34] J.-M. Schmidt and M. Hutter, *Optical and em fault-attacks on crt-based rsa: Concrete results*. na, 2007.

[35] 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 17)*. USENIX Association, 2017.

[36] J. Obermaier and S. Tatschner, "Shedding too much light on a microcontrollers firmware protection," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX Association, 2017.

[37] G. Bouffard, J. Iguchi-Cartigny, and J.-L. Lanet, "Combined software and hardware attacks on the java card control flow." in *CARDIS*, vol. 7079. Springer, 2011, pp. 283–296.

[38] G. Barbu, H. Thiebeauld, and V. Guerin, "Attacks on java card 3.0 combining fault and logical attacks," *Smart Card Research and Advanced Application*, pp. 148–163, 2010.

[39] J. Lancia, "Java card combined attacks with localization-agnostic fault injection," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2012, pp. 31–45.

[40] E. Vétillard and A. Ferrari, "Combined attacks and countermeasures," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2010, pp. 133–147.

[41] F. Amiel, K. Villegas, B. Feix, and L. Marcel, "Passive and active combined attacks: Combining fault attacks and side channel analysis," in *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on*. IEEE, 2007, pp. 92–102.

[42] C. Clavier, B. Feix, G. Gagnerot, and M. Roussellet, "Passive and active combined attacks on aes combining fault attacks and side channel analysis," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*. IEEE, 2010, pp. 10–19.

[43] A. Moradi, O. Mischke, C. Paar, Y. Li, K. Ohta, and K. Sakiyama, "On the power of fault sensitivity analysis and collision side-channel attacks in a combined setting." in *CHES*, vol. 6917. Springer, 2011, pp. 292–311.

[44] T. Roche, V. Lomné, and K. Khalfallah, "Combined fault and side-channel attack on protected implementations of aes," *Smart Card Research and Advanced Applications*, pp. 65–83, 2011.

[45] M. Joye, "A method for preventing "skipping" attacks," in *2012 IEEE Symposium on Security and Privacy Workshops, San Francisco, CA, USA, May 24-25, 2012*, 2012, pp. 12–15. [Online]. Available: http://dx.doi.org/10.1109/SPW.2012.14

[46] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson, "Formal verification of a software countermeasure against instruction skip attacks," Cryptology ePrint Archive, Report 2013/679, 2013, http://eprint.iacr.org/.

[47] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*. Springer, 2006.

[48] N. Theißing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, "Comprehensive analysis of software countermeasures against fault attacks," in *Proc. of DATE'13*, 2013, pp. 404–409.

[49] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, "Countermeasures against fault attacks on software implemented AES: effectiveness and cost," in *Proc of WESS'10*, 2010, pp. 1–10.

[50] J.-F. Lalande, K. Heydemann, and P. Berthomé, "Software countermeasures

for control flow integrity of smart card c codes," in *European Symposium on Research in Computer Security*. Springer, 2014, pp. 200–218.

[51] S. Patranabis, A. Chakraborty, and D. Mukhopadhyay, "Fault tolerant infective countermeasure for aes," *Journal of Hardware and Systems Security*, vol. 1, no. 1, pp. 3–17, 2017.

[52] I. Koren and C. M. Krishna, *Fault-tolerant systems*. Morgan Kaufmann, 2010.

[53] B. Gierlichs, J.-M. Schmidt, and M. Tunstall, "Infective computation and dummy rounds: Fault protection for block ciphers without check-before-output."

[54] D. P. Siewiorek, "Architecture of fault-tolerant computers: an historical perspective," *Proceedings of the IEEE*, vol. 79, no. 12, pp. 1710–1734, Dec 1991.

[55] R. P. Bastos, F. S. Torres, J. M. Dutertre, M. L. Flottes, G. D. Natale, and B. Rouzeyre, "A bulk built-in sensor for detection of fault attacks," in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST 2013)*, 2013, pp. 51–54.

[56] A. G. Yanci, S. Pickles, and T. Arslan, "Detecting voltage glitch attacks on secure devices," in *ECSIS Symposium on Bio-inspired Learning and Intelligent Systems for Security (BLISS 2008)*. IEEE, 2008, pp. 75–80.

[57] E. Garcia-Moreno, R. Picos, E. Isern, M. Roca, J. Font, and K. Suenaga, "Cmos current source based radiation sensors," in *IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT 2010)*, 2010, pp. 1380–1383.

[58] P. Luo, C. Luo, and Y. Fei, "System clock and power supply cross-checking for glitch detection," *IACR Cryptology ePrint Archive*, vol. 2016, p. 968, 2016. [Online]. Available: http://eprint.iacr.org/2016/968

[59] S. Das, D. Roberts, S. Lee, S. Pant, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "A self-tuning dvs processor using delay-error detection and correction," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 4, pp. 792–804, April 2006.

[60] D. Mukhopadhyay, "An improved fault based attack of the advanced encryption standard," in *International Conference on Cryptology in Africa (AFRICACRYPT)*, 2009, pp. 421–434. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02384-2_26

[61] M. Tunstall, D. Mukhopadhyay, and S. Ali, "Differential fault analysis of the advanced encryption standard using a single fault," in *IFIP WG 11.2 International Workshop Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication (WISTP 2011)*, 2011, pp. 224–233. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21040-2_15

[62] C. Ferretti, S. Mella, and F. Melzani, "The role of the fault model in DFA against AES," in *Proc. of HASP'14*, 2014, pp. 4:1–4:8.

[63] D. Karaklajic, J. Schmidt, and I. Verbauwhede, "Hardware Designer's Guide to Fault Attacks," *IEEE Trans. VLSI Syst.*, vol. 21, no. 12, pp. 2295–2306, 2013.

[64] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks

on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, Nov 2012.

[65] M. Otto, "Fault attacks and countermeasures," Ph.D. dissertation, University of Paderborn, 2005.

[66] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, "Countermeasures Against Fault Attacks on Software Implemented AES: Effectiveness and Cost," in *Proc. of WESS'10*, 2010, pp. 7:1–7:10.

[67] B. Selmke, J. Heyszl, and G. Sigl, "Attack on a DFA protected AES by simultaneous laser fault injections," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2016)*, 2016, pp. 36–46. [Online]. Available: http://dx.doi.org/10.1109/FDTC.2016.16

[68] S. Patranabis, A. Chakraborty, D. Mukhopadhyay, and P. P. Chakrabarti, "Fault space transformation: A generic approach to counter differential fault analysis and differential fault intensity analysis on aes-like block ciphers," *IEEE Trans. Information Forensics and Security*, vol. 12, no. 5, pp. 1092–1102, 2017. [Online]. Available: http://dx.doi.org/10.1109/TIFS.2016.2646638

[69] E. Ozer, Y. Sazeides, D. Kershaw, and S. Biles, "Data processing apparatus and method for analysing transient faults withing storage elements of the data processing apparatus," Apr. 25 2013, US Patent 2013/0103972 A1.

[70] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Moderator-Ravi, "Security as a New Dimension in Embedded System Design," in *Proc. of the DAC'04*, 2004, pp. 753–760.

[71] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM TECS*, vol. 3, no. 3, pp. 461–491, 2004.

[72] B. Robisson, M. Agoyan, P. Soquet, S. Le Henaff, F. Wajsbürt, P. Bazargan-Sabet, and G. Phan, "Smart security management in secure devices," Cryptology ePrint Archive, Report 2015/670, 2015. http://eprint. iacr. org, Tech. Rep., 2015.

[73] G. Gogniat, T. Wolf, W. Burleson, J.-P. Diguet, L. Bossuet, and R. Vaslin, "Reconfigurable hardware for high-security/high-performance embedded systems: the safes perspective," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 2, pp. 144–155, 2008.

[74] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[75] B. Yuce, N. F. Ghalaty, C. Deshpande, C. Patrick, L. Nazhandali, and P. Schaumont, "FAME: Fault-attack aware microprocessor extensions for hardware fault detection and software fault response," in *Proc. of HASP'16*, 2016, p. 8.

[76] F. Courbon, P. Loubet-Moundi, J. J. Fournier, and A. Tria, "Adjusting laser injections for fully controlled faults," in *Proc. of COSADE'14*, 2014, pp. 229–242.

[77] Y. Li, K. Sakiyama, S. Gomisawa, T. Fukunaga, J. Takahashi, and K. Ohta, "Fault Sensitivity Analysis," in *Proc. of CHES'10*, 2010, pp. 320–334.

[78] M. Joye and M. Tunstall, Eds., *Fault Analysis in Cryptography*, ser. Information Security and Cryptography. Springer, 2012.

[79] S. Bhattacharya and D. Mukhopadhyay, "Formal fault analysis of branch predictors: attacking countermeasures of asymmetric key ciphers," *Journal of Cryptographic Engineering*, vol. 7, no. 4, pp. 299–310, 2017.

[80] S. Guilley, L. Sauvage, J.-L. Danger, N. Selmane, and R. Pacalet, "Silicon-level solutions to counteract passive and active attacks," in *Fault Diagnosis and Tolerance in Cryptography, 2008. FDTC'08. 5th Workshop on.* IEEE, 2008, pp. 3–17.

[81] L. Zussa, J.-M. Dutertre, J. Clédiere, B. Robisson, A. Tria *et al.*, "Investigation of timing constraints violation as a fault injection means," in *Proc. of DCIS'12*, 2012.

[82] "Riscure Inspector FI," https://www.riscure.com/security-tools/inspector-fi/, [Online; accessed 18-May-2017].

[83] M. Hutter and J.-M. Schmidt, "The temperature side channel and heating fault attacks," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2013, pp. 219–235.

[84] S. Skorobogatov, "Local heating attacks on flash memory devices," in *Hardware-Oriented Security and Trust, 2009. HOST'09. IEEE International Workshop on.* IEEE, 2009, pp. 1–6.

[85] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine," in *Security and Privacy, 2003. Proceedings. 2003 Symposium on.* IEEE, 2003, pp. 154–165.

[86] T. Korak, M. Hutter, B. Ege, and L. Batina, "Clock glitch attacks in the presence of heating," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014 Workshop on.* IEEE, 2014, pp. 104–114.

[87] S. P. Skorobogatov, R. J. Anderson *et al.*, "Optical fault induction attacks," in *CHES*, vol. 2523. Springer, 2002, pp. 2–12.

[88] P. Maistri, R. Leveugle, L. Bossuet, A. Aubert, V. Fischer, B. Robisson, N. Moro, P. Maurine, J.-M. Dutertre, and M. Lisart, "Electromagnetic analysis and fault injection onto secure circuits," in *Very Large Scale Integration (VLSI-SoC), 2014 22nd International Conference on.* IEEE, 2014, pp. 1–6.

[89] R. V. S. R. Velegalati and J. van Woudenberg, "Electro magnetic fault injection in practice," in *International Cryptographic Module Conference (ICMC)*, 2013.

[90] Y. Cai, S. Ghose, Y. Luo, K. Mai, O. Mutlu, and E. F. Haratsch, "Vulnerabilities in mlc nand flash memory programming: experimental analysis, exploits, and mitigation techniques," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on.* IEEE, 2017, pp. 49–60.

[91] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 361–372.

[92] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack." in *USENIX Security Symposium*, 2016, pp. 1–18.

[93] A. Kurmus, N. Ioannou, N. Papandreou, and T. Parnell, "From random block corruption to privilege escalation: A filesystem attack vector for rowhammer-like attacks," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/woot17/workshop-program/presentation/kurmus

[94] A. Dehbaoui, A.-P. Mirbaha, N. Moro, J.-M. Dutertre, and A. Tria, "Electromagnetic glitch on the aes round counter," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2013, pp. 17–31.

[95] S. Nashimoto, N. Homma, Y.-i. Hayashi, J. Takahashi, H. Fuji, and T. Aoki, "Buffer overflow attack with multiple fault injection and a proven countermeasure," *Journal of Cryptographic Engineering*, vol. 7, no. 1, pp. 35–46, 2017.

[96] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil, "Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 213–222.

[97] H. Choukri and M. Tunstall, "Round reduction using faults," *FDTC*, vol. 5, pp. 13–24, 2005.

[98] J.-M. Dutertre, A.-P. Mirbaha, D. Naccache, A.-L. Ribotta, A. Tria, and T. Vaschalde, "Fault round modification analysis of the advanced encryption standard," in *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 140–145.

[99] E. Biham and A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," in *Advances in CryptologyCRYPTO'97*. Springer, 1997, pp. 513–525.

[100] J. J. Hoch and A. Shamir, "Fault analysis of stream ciphers," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2004, pp. 240–253.

[101] I. Biehl, B. Meyer, and V. Müller, "Differential fault attacks on elliptic curve cryptosystems," in *Annual International Cryptology Conference*. Springer, 2000, pp. 131–146.

[102] M. Taha and T. Eisenbarth, "Implementation attacks on post-quantum cryptographic schemes," Cryptology ePrint Archive, Report 2015/1083, 2015, http://eprint.iacr.org/.

[103] S.-M. Yen and M. Joye, "Checking before output may not be enough against fault-based cryptanalysis," *IEEE Transactions on computers*, vol. 49, no. 9, pp. 967–970, 2000.

[104] J. Blömer and J.-P. Seifert, "Fault based cryptanalysis of the advanced encryption standard (aes)," in *Computer Aided Verification*. Springer, 2003, pp. 162–181.

[105] M. Ciet and M. Joye, "Elliptic curve cryptosystems in the presence of permanent and transient faults," *Designs, codes and cryptography*, vol. 36, no. 1, pp. 33–43, 2005.

[106] P.-A. Fouque, R. Lercier, D. Réal, and F. Valette, "Fault attack on elliptic curve montgomery ladder implementation," in *Fault Diagnosis and Tolerance in Cryptography, 2008. FDTC'08. 5th Workshop on*. IEEE, 2008, pp. 92–98.

[107] N. F. Ghalaty, B. Yuce, M. Taha, and P. Schaumont, "Differential fault intensity analysis," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014 Workshop on*. IEEE, 2014, pp. 49–58.

[108] 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, no. 1, pp. 88–97, 2012.

[109] Y. Liu, J. Zhang, L. Wei, F. Yuan, and Q. Xu, "Dera: Yet another differential fault attack on cryptographic devices based on error rate analysis," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 31.

[110] T. Fuhr, E. Jaulmes, V. Lomné, and A. Thillard, "Fault attacks on aes with faulty ciphertexts only," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*. IEEE, 2013, pp. 108–118.

[111] K. Järvinen, C. Blondeau, D. Page, and M. Tunstall, "Harnessing biased faults in attacks on ecc-based signature schemes," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2012 Workshop on*. IEEE, 2012, pp. 72–82.

[112] A. Vasselle, H. Thiebeauld, Q. Maouhoub, A. Morisset, and S. Ermeneux, "Laser-induced fault injection on smartphone bypassing the secure boot," in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2017, pp. 41–48.

[113] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," *Black Hat*, 2015.

[114] M. San Pedro, M. Soos, and S. Guilley, "Fire: Fault injection for reverse engineering." in *WISTP*. Springer, 2011, pp. 280–293.

[115] H. Le Bouder, S. Guilley, B. Robisson, and A. Tria, "Fault injection to reverse engineer des-like cryptosystems," in *Foundations and Practice of Security*. Springer, 2014, pp. 105–121.

[116] C. Clavier and A. Wurcker, "Reverse engineering of a secret aes-like cipher by ineffective fault analysis," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*. IEEE, 2013, pp. 119–128.

[117] M. Jacob, D. Boneh, and E. Felten, "Attacking an obfuscated cipher by injecting faults," in *Digital Rights Management Workshop*, vol. 2696. Springer, 2002, pp. 16–31.

[118] F. Courbon, J. J. Fournier, P. Loubet-Moundi, and A. Tria, "Combining image processing and laser fault injections for characterizing a hardware aes," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 34, no. 6, pp. 928–936, 2015.

[119] P. E. Dodd and L. W. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 583–602, June 2003.

[120] E. Sho, N. Homma, Y.-i. Hayashi, J. Takahashi, and F. Hitoshi, "An adaptive multiple-fault injection attack on microcontrollers and a countermeasure," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 98, no. 1, pp. 171–181, 2015.

[121] "LEON3 processor," http://www.gaisler.com/index.php/products/processors/leon3, [Online; accessed 18-May-2017].

[122] 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*. Springer, 2010, pp. 182–193.

[123] "SAKURA-G Board," http://satoh.cs.uec.ac.jp/SAKURA/hardware/SAKURA, [Online; accessed 18-May-2015].

[124] S. Endo, T. Sugawara, N. Homma, T. Aoki, and A. Satoh, "An On-chip Glitchy-clock Generator for Testing Fault Injection Attacks," *Journal of Cryptographic Engineering*, vol. 1, no. 4, pp. 265–270, 2011.

[125] "GRLIB IP library," http://www.gaisler.com/index.php/products/ipcores/soclibrary, [Online; accessed 18-May-2015].

[126] "GRMON2 Debug Monitor," http://www.gaisler.com/index.php/products/debug-tools/grmon2, [Online; accessed 18-May-2015].

[127] Y. Li, K. Ohta, and K. Sakiyama, "Revisit Fault Sensitivity Analysis on WDDL-AES ," in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST),*. IEEE, 2011, pp. 148–153.

[128] R. Lashermes, G. Reymond, J. Dutertre, J. Fournier, B. Robisson, and A. Tria, "A DFA on AES Based on the Entropy of Error Distributions," in *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC),*. IEEE, 2012, pp. 34–43.

[129] B. Yuce, N. F. Ghalaty, and P. Schaumont, "Tvvf: Estimating the vulnerability of hardware cryptosystems against timing violation attacks," in *Hardware*

Oriented Security and Trust (HOST), 2015 IEEE International Symposium on. IEEE, 2015, pp. 72–77.

[130] T. Sugawara, D. Suzuki, and T. Katashita, "Circuit Simulation for Fault Sensitivity Analysis and its Application to Cryptographic LSI," in *Proc. of FDTC'12*, 2012, pp. 16–23.

[131] A. Barenghi, L. Breveglieri, A. Palomba, and G. Pelosi, "Fault Sensitivity Analysis at Design Time," in *Trusted Computing for Embedded Systems*. Springer, 2015, pp. 175–186.

[132] Y.-i. Hayashi, N. Homma, T. Mizuki, T. Aoki, and H. Sone, "Fundamental study on fault occurrence mechanisms by intentional electromagnetic interference using impulses," in *Proc. of APEMC'15*, 2015, pp. 585–588.

[133] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[134] G. Piret and J. Quisquater, "A differential fault attack technique against SPN structures, with application to the AES and KHAZAD," in *Proc. of CHES'03*, 2003, pp. 77–88.

[135] D. Ferraretto and G. Pravadelli, "Simulation-based fault injection with qemu for speeding-up dependability analysis of embedded software," *Journal of Electronic Testing*, vol. 32, no. 1, pp. 43–57, 2016.

[136] A. Höller, G. Schönfelder, N. Kajtazovic, T. Rauter, and C. Kreiner, "Fies: a fault injection framework for the evaluation of self-tests for cots-based safety-critical systems," in *Microprocessor Test and Verification Workshop (MTV), 2014 15th International*. IEEE, 2014, pp. 105–110.

[137] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, and D. Gizopoulos, "Differential fault injection on microarchitectural simulators," in *Workload Characterization (IISWC), 2015 IEEE International Symposium on.* IEEE, 2015, pp. 172–182.

[138] M. Kooli and G. Di Natale, "A survey on simulation-based fault injection tools for complex systems," in *Design & Technology of Integrated Systems In Nanoscale Era (DTIS), 2014 9th IEEE International Conference On.* IEEE, 2014, pp. 1–6.

[139] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas, "Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on.* IEEE, 2014, pp. 622–629.

[140] H. Schirmeier, C. Borchert, and O. Spinczyk, "Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors," in *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on.* IEEE, 2015, pp. 319–330.

[141] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk, "Fail*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance," in *Dependable Computing Conference (EDCC), 2015 Eleventh European.* IEEE, 2015, pp. 245–255.

[142] M. Berthier, J. Bringer, H. Chabanne, T.-H. Le, L. Rivière, and V. Servant, "Idea: embedded fault injection simulator on smartcard," in *International*

Symposium on Engineering Secure Software and Systems. Springer, 2014, pp. 222–229.

[143] P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky, and J.-F. Lalande, "High level model of control flow attacks for smart card functional security," in *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*. IEEE, 2012, pp. 224–229.

[144] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. de Choudens, "FISSC: A fault injection and simulation secure collection," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2016, pp. 3–11.

[145] L. Dureuil, M.-L. Potet, P. de Choudens, C. Dumas, and J. Clédière, "From code review to fault injection attacks: filling the gap using fault model inference," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2015, pp. 107–124.

[146] L. Rivière, M.-L. Potet, T.-H. Le, J. Bringer, H. Chabanne, and M. Puys, "Combining high-level and low-level approaches to evaluate software implementations robustness against multiple fault injection attacks," in *International Symposium on Foundations and Practice of Security*. Springer, 2014, pp. 92–111.

[147] A. Höller, A. Krieg, T. Rauter, J. Iber, and C. Kreiner, "Qemu-based fault injection for a system-level analysis of software countermeasures against fault attacks," in *Digital System Design (DSD), 2015 Euromicro Conference on*. IEEE, 2015, pp. 530–533.

[148] M. Lackner, R. Berlach, M. Hraschan, R. Weiss, and C. Steger, "A fault attack emulation environment to evaluate java card virtual-machine security," in *Digital System Design (DSD), 2014 17th Euromicro Conference on*. IEEE, 2014, pp. 480–487.

[149] J.-B. Machemie, C. Mazin, J.-L. Lanet, and J. Cartigny, "Smartcm a smart card fault injection simulator," in *Information Forensics and Security (WIFS), 2011 IEEE International Workshop on*. IEEE, 2011, pp. 1–6.

[150] M. Puys, L. Rivière, J. Bringer, and T.-h. Le, "High-level simulation for multiple fault injection evaluation," in *Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance*. Springer, 2015, pp. 293–308.

[151] L. Rivière, J. Bringer, T.-H. Le, and H. Chabanne, "A novel simulation approach for fault injection resistance evaluation on smart cards," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*. IEEE, 2015, pp. 1–8.

[152] J. Breier, "On Analyzing Program Behavior Under Fault Injection Attacks," Cryptology ePrint Archive, Report 2016/1060, 2016, http://eprint.iacr.org/.

[153] N. Selmane, S. Bhasin, S. Guilley, T. Graba, and J.-L. Danger, "WDDL is protected against setup time violation attacks," in *Proc. of FDTC'09*, 2009, pp. 73–83.

[154] L. Zussa, A. Dehbaoui, K. Tobich, J.-M. Dutertre, P. Maurine, L. Guillaume-Sage, J. Clediere, and A. Tria, "Efficiency of a glitch detector against electromagnetic fault injection," in *Proc. of DATE'14*, 2014, pp. 1–6.

[155] X. Guo, D. Mukhopadhyay, C. Jin, and R. Karri, "Security analysis of concurrent error detection against differential fault analysis," *Journal of Cryptographic Engineering*, pp. 1–17, 2014.

[156] T. Sato and Y. Kunitake, "A simple flip-flop circuit for typical-case designs for DFM," in *Proc. of ISQED'07*, 2007, pp. 539–544.

[157] S. Gregori, A. Cabrini, O. Khouri, and G. Torelli, "On-chip error correcting techniques for new-generation flash memories," *Proceedings of the IEEE*, vol. 91, no. 4, pp. 602–616, 2003.

[158] M. Hutter and P. Schwabe, "Nacl on 8-bit AVR microcontrollers," in *International Conference on Cryptology in Africa (AFRICACRYPT 2013)*, 2013, pp. 156–172. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38553-7_9

[159] K. Lemke-Rust and C. Paar, "An adversarial model for fault analysis against low-cost cryptographic devices." in *FDTC*. Springer, 2006, pp. 131–143.

[160] "AMBA Bus Specifications," https://www.arm.com/products/system-ip/amba-specifications, [Online; accessed 18-May-2017].

[161] "GRLIB IP Core User's Manual," http://www.gaisler.com/products/grlib/grip.pdf, [Online; accessed 18-May-2015].

[162] C. Deshpande, "Hardware fault attack detection methods for secure embedded systems," Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, 9 2017.

[163] M. Taha, A. Reyhani-Masoleh, and P. Schaumont, "Stateless leakage resiliency from nlfsrs," in *Hardware Oriented Security and Trust (HOST), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 56–61.

[164] S. Ordas, L. Guillaume-Sage, and P. Maurine, "Em injection: Fault model and locality," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2015 Workshop on*. IEEE, 2015, pp. 3–13.

[165] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey, "Software-implemented EDAC protection against SEUs," *IEEE Trans. Reliability*, vol. 49, no. 3, pp. 273–284, 2000.

[166] M. Witteman and M. Oostdijk, "Secure application programming in the presence of side channel attacks," in *RSA conference*, vol. 2008, 2008.

[167] S. S. Ali, D. Mukhopadhyay, and M. Tunstall, "Differential fault analysis of AES: towards reaching its limits," *Journal of Cryptographic Engineering*, vol. 3, no. 2, pp. 73–97, 2013. [Online]. Available: http://dx.doi.org/10.1007/s13389-012-0046-y

[168] A.-P. Mirbaha, J.-M. Dutertre, and A. Tria, "Differential analysis of round-reduced AES faulty ciphertexts," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT 2013)*. IEEE, 2013, pp. 204–211.

[169] X. Guo, D. Mukhopadhyay, C. Jin, and R. Karri, "Security analysis of concurrent error detection against differential fault analysis," *Journal of Cryptographic Engineering*, vol. 5, no. 3, pp. 153–169, 2015.

[170] E. Barker and J. Kelsey, "Recommendation for random number generation using deterministic random bit generators," NIST Special Publication 800-90A rev 1, June 2015.

[171] B. Yuce, N. F. Ghalaty, C. Deshpande, H. Santapuri, C. Patrick, L. Nazhandali, and P. Schaumont, "Analyzing the fault injection sensitivity of secure embedded software," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 4, p. 95, 2017.

[172] N. F. Ghalaty, B. Yuce, and P. Schaumont, "Analyzing the efficiency of biased-fault based attacks," *IEEE Embedded Systems Letters*, vol. 8, no. 2, pp. 33–36, 2016.

[173] N. F. Ghalaty, B. Yuce, and P. Schaumont, "Differential fault intensity analysis on PRESENT and LED block ciphers," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2015, pp. 174–188.

[174] C. Patrick, B. Yuce, N. F. Ghalaty, and P. Schaumont, "Lightweight fault attack resistance in software using intra-instruction redundancy," in *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, 2016, pp. 231–244. [Online]. Available: https://doi.org/10.1007/978-3-319-69453-5_13

[175] C. Deshpande, B. Yuce, N. F. Ghalaty, D. Ganta, P. Schaumont, and L. Nazhandali, "A configurable and lightweight timing monitor for fault attack detection," in *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*. IEEE, 2016, pp. 461–466.

[176] N. F. Galathy, B. Yuce, and P. Schaumont, "A systematic approach to fault attack resistant design," in *Fundamentals of IP and SoC Security*. Springer, 2017, pp. 223–245.

[177] B. Yuce and S. P. R. Ghalaty, Nahid Farhady and, "Microprocessor fault detection and response system," Nov. 17 2017, US Patent 20,170,344,438. [Online]. Available: http://www.freepatentsonline.com/y2017/0344438.html