Fault Attacks on Cryptosystems: Novel Threat Models, Countermeasures and Evaluation Metrics

Nahid Farhady Ghalaty

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 Schaumont, Chair

Chao Wang

Leyla Nazhandali

Daphen Yao

Yaling Yang

Lynn Abbott

July 26, 2016

Blacksburg, Virginia

Keywords: Hardware Security, Physical Attacks, Cryptography

Copyright 2016, Nahid Farhady Ghalaty

Fault Attacks on Cryptosystems: Novel Threat Models, Countermeasures and Evaluation Metrics

Nahid Farhady Ghalaty

(ABSTRACT)

Recent research has demonstrated that there is no sharp distinction between passive attacks based on side-channel leakage and active attacks based on fault injection. Fault behavior can be processed as side-channel information, offering all the benefits of Differential Power Analysis including noise averaging and hypothesis testing by correlation. In fault attacks, the adversary induces faults into a device while it is executing a known program and observes the reaction. The abnormal reactions of the device are later analyzed to obtain the secrets of the program under execution.

Fault attacks are a powerful threat. They are used to break cryptosystems, Pay TVs, smart cards and other embedded applications. In fault attack resistant design, the fault is assumed to be induced by a smart, malicious, determined attacker who has high knowledge of the design under attack. Moreover, the purpose of fault attack resistant design is for the system to work correctly under intentional fault injection without leaking any secret data information.

Towards building a fault attack resistant design, the problem can be categorized into three main subjects:

- Investigating novel and more powerful threat models and attack procedures.
- Proposing countermeasures to build secure systems against fault attacks
- Building evaluation metrics to measure the security of designs

In this regard, my thesis has covered the first bullet, by proposing the Differential Fault Intensity Analysis (DFIA) based on the biased fault model. The biased fault model in this attack means the gradual behavior of the fault as a cause of increasing the intensity of fault injection. The DFIA attack has been successfully launched on AES, PRESENT and LED block ciphers. Our group has also recently proposed this attack on the AES algorithm running on a LEON3 processor.

In our work, we also propose a countermeasure against one of the most powerful types of fault attacks, namely, Fault Sensitivity Analysis (FSA). This countermeasure is based on balancing the delay of the circuit to destroy the correlation of secret data and timing delay of a circuit.

Additionally, we propose a framework for assessing the vulnerability of designs against fault attacks. An example of this framework is the Timing Violation Vulnerability Factor (TVVF) that is a metric for measuring the vulnerability of hardware against timing violation attacks. We compute TVVF for two implementations of AES algorithm and measure the vulnerability of these designs against two types of fault attacks.

As shown in this thesis, fault attacks are more serious threat than considered by the cryptography community. This thesis provides a deep understanding of the fault behavior in the

circuit and therefore a better knowledge on powerful fault attacks. The techniques developed in this dissertation focus on different aspects of fault attacks on hardware architectures and microprocessors. Considering the proposed fault models, attacks, and evaluation metrics in this thesis, there is hope to develop robust and fault attack resistant microprocessors. We conclude this thesis by observing future areas and opportunities for research.

Dedication

This thesis is proudly dedicated to

My parents, Nasrin Parizad Ghalati & Keramatollah Farhadi Ghalati,

And my sister, Nastaran Farhadi Ghalati

without whom none of my success would be possible.

Acknowledgments

I would first like to thank my thesis advisor Dr. Schaumont of the Bradley Department of Electrical and Computer Engineering at Virginia Tech. The door to Dr. Schaumont's office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this thesis to be my own work, but steered me in the right the direction whenever he thought I needed it. Not only he has been my PhD. advisor, I would like to express my highest gratitude to him for teaching me life lessons in encountering hardship and difficulty.

I would also like to thank my PhD. committee members, for their feedback and support for this research project: Dr. Abbott, Dr. Nazhandali, Dr. Wang, Dr. Yao and Dr. Yang. Without their passionate participation and input, this thesis could not have been successfully conducted.

I would like to gratefully thank the National Science Foundation and Semiconductor Research Corporation for supporting this project through Grant 1441710, Grant 1115839. I would also like to acknowledge the Bradley Department of Electrical and Computer Engineering,

Virginia Tech for their financial support.

I thank my fellow labmates in Secure Embedded Systems Lab: Bilgiday Yuce, Aydin Aysu, Mostafa Taha, Chinmay Deshpande, Conor Patrick, Moein Pahlavan Yaki, Krishna Pabbuleti, Deepak Mane, Ege Gulcan and so many other supporting fellows for enlightening me the first glance of research.

Finally, I must express my very profound gratitude to my parents and to my sister for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. I would like to thank so many beloved friends who surrounded me with love, advice and prayers while I was far away from my family. Thanks to Erfan, Sepideh, Mehrnoosh, Nasibeh, Donia and Saloumeh for being there for me. This accomplishment would have not been possible without them. Thank you.

Contents

1 Introduction **1**

2 Background **8**

2.1 Fault Attack 8

2.2 Fundamentals of Fault Attacks 9

2.2.1 Fault Model 10

2.2.2 Fault Injection Techniques 11

2.3 Fault Attack Procedure 13

2.3.1 Fault Measurement 14

2.3.2 Fault Measurement Steps 15

2.3.3 Fault Measurement Case Study 18

2.3.4 Fault Analysis: Differential Fault Attack 20

3 Biased Fault Models **23**

3.1 Challenges of Traditional Fault Attacks 23

3.2 Clock Glitch Injection Mechanisms 25

3.2.1 Setup Time Violation 25

3.2.2 Causes of Fault Bias 26

3.2.3 Quantifying Fault Bias 29

3.2.4 Effects of Operating Conditions and Data on Fault Bias 31

4 Differential Fault Intensity Analysis (DFIA) **34**

4.1 Fault Attack Framework 34

4.2 Differential Fault Intensity Analysis 36

4.3 DFIA on Symmetric Key Cryptographic Primitives 39

4.3.1 DFIA on AES 39

4.3.2 Methodology for Selecting Fault Set 42

4.3.3 Experimental Setup 44

4.3.4 Biased Fault Experiment for S-Box Architectures 46

4.3.5 DFIA Experiment on AES 48

4.3.6 AES Key Retrieval Process with One Byte Fault Injection 52

4.3.7 AES Key Retrieval Process with Multiple Byte Fault Injection 55

4.3.8 DFIA on PRESENT and LED 56

4.3.9 PRESENT Block Cipher 56

4.3.10 LED Block Cipher 59

4.3.11 Implementations of the Block Ciphers 60

4.3.12 Biased Fault Injection in PRESENT and LED 61

4.3.13 Biased Faults in PRESENT and LED Exist 62

4.3.14 Post-Processing of DFIA on PRESENT 63

4.3.15 Post-processing of DFIA on LED 65

4.3.16 Results of DFIA on PRESENT and LED 67

4.4 DFIA with Multiple Plaintexts 69

4.4.1 Trade-off between Fault Injection Resolution and Number of Plaintexts 71

5 Analyzing the Efficiency of Biased-Fault Based Attacks 74

5.1 Effects of Fault Bias on Circuit Behavior 75

5.2 Biased Based Fault Attacks 80

5.2.1 Fault Sensitivity Analysis (FSA) 81

5.2.2 Non-Uniform Error Value Analysis (NUEVA) 81

5.2.3 Non-Uniform Faulty Value Attack (NUFVA) 82

5.2.4 Differential Fault Intensity Analysis (DFIA) 83

5.3 Experimental Setup 84

5.4 Efficiency Analysis of Biased Based Fault Attacks 86

5.4.1 Results for Ideal Fault Injection 86

5.4.2 Results for Noisy Fault Injection 92

5.4.3 Attack Efficiency 93

5.5 Conclusion 94

6 A Design-Time Countermeasure against Fault-Sensitivity Analysis 96

6.1 Related Work 97

6.2 Fault Sensitivity Analysis 98

6.3 Data Dependency of Fault Sensitivity on S-box Architectures 100

6.3.1 PPRM1 S-box 100

6.3.2 Boyar-Peralta S-box 101

6.4 Data Analysis 103

6.5 Proposed Countermeasure 106

6.5.1 FSA Resistant PPRM1 design 107

6.5.2 FSA Resistant AES 108

6.6 Experimental Setup 109

7 A Design Metric to Quantify Fault Attack Sensitivity 112

7.1 A Design Metric for Fault Attack Evaluation 113

7.2 Timing Violation Vulnerability Factor 115

7.2.1 Timing characterization of the SoV 118

7.2.2 How do we compute TVVF for an attack on a circuit? 123

7.3 Results 126

7.3.1 Experimental Verification of Fault Generation Probability 128

7.3.2 An Example Case Study 129

7.4 Related Work 132

8 Conclusion 133

8.1 Overall Conclusion and Future Work 137

Bibliography 139

List of Figures

2.1 A Fault Attack Requires Fault Measurement and Fault Analysis 15

2.2 Sample Combinational Logic 18

2.3 AES Structure 21

3.1 The effect of the glitch injection on the clock signal 25

3.2 **(a)** A block diagram of a 4-bit ripple-carry adder implemented on an FPGA. **(b)** The distribution of path delays within fan-in cone of each output bit. Using the non-uniform path delay distribution, we can create biased faults (e.g, 1-bit, 2-bit, etc.) 27

3.3 Number of violated paths with respect to the applied fault intensity and the corresponding glitch period for the ripple-carry adder. 29

3.4 The effect of varying the supply voltage and the operating temparature on the path delays. Increasing the temperature and decreasing the supply voltage increase the path delays. 32

3.5 Illustration of data-dependency of paths delays on the ripple-carry adder: **(a)** $T_{Q3} > T_{Q2} > T_{Q1} > T_{Q0}$. **(b)** $T_{Q3} > T_{Q1} > T_{Q2} > T_{Q0}$. 33

4.1 Generic Fault Attack Framework 35

4.2 Effect of Choosing Fault Set on Key Recovery Convergence 41

4.3 Block Diagram of the Experimental Setup 45

4.4 DUT Architecture for the Biased Fault Experiment 47

4.5 A High-level Timing Diagram for the Biased Fault Experiment 48

4.6 Biased Fault Behavior for Different S-box Implementations 49

4.7 (a) Fault-injection Flip-flop (FI-FF) Architecture. (b) The DUT Architecture for Fault Injection into k-th Byte of the AES State. 50

4.8 A High-level Timing Diagram for FI-FF 51

4.9 The Key Hypothesis Testing Algorithm Demonstration on a Measured Result, Step 1 and 2 54

4.10 The Key Hypothesis Testing Algorithm Demonstration on a Measured Result, Step 3 55

4.11 Nibble-serial Implementation of PRESENT 58

4.12 Nibble-serial Implementation of LED 59

4.13 (a)Block Diagram of Experimental Setup (b)Timing Diagram of Experimental Setup 61

4.14 Biased Fault in the PRESENT and LED Implementations 62

4.15 Biased Fault Injection on State of (a) PRESENT and (b) LED 63

4.16 DFIA Steps to Retrieve 4-bit Key of PRESENT 65

4.17 Trade-off Between Fault Injection Resolution and Number of Plaintexts used for (a)PRESENT and (b)LED 72

5.1 The relationship between the critical timing delay and the Hamming weight of the input of PPRM1-SBOX [1]. 76

5.2 The distribution of injected error values in the output of PPRM1-SBOX ($Fault\ Intensity = 0.238GHz$). 77

5.3 The distribution of faulty values in the output of Comp-SBOX ($FaultIntensity = 0.250GHz$). 78

5.4 The relationship between the critical timing delay and the Hamming weight of the input of PPRM1-SBOX [2] 80

5.5 **(a)** Block diagram for the experimental setup. **(b)** Timing diagram for the experimental setup. 85

5.6 Number of Required Fault Injection Attempts to Retrieve the Key with Different Attack Strategies in Ideal Condition (a)(b)with Fault Sensitivity Information 87

5.7 Number of Required Fault Injection Attempts to Retrieve the Key with Different Attack Strategies in Ideal Condition (a)(b)with All Possible Fault Intensities 88

5.8 Number of Required Fault Injection Attempts to Retrieve the Key with Different Attack Strategies in Noisy Environment. Each group shows the rounds that are affected by the fault injection. 90

5.9 Number of Required Fault Injection Attempts to Retrieve one Byte of the Key with Different Attack Strategies in Noisy Environment. Each group shows the rounds that are affected by the fault injection. 91

6.1 Data Dependency of Fault Sensitivity for Different Gates [3] 99

6.2 Data Dependency of Fault Sensitivity with Input Hamming Weight in PPRM1 S-box (Note the Y-scale Range) 101

6.3 Data Dependency of Fault Sensitivity with Input Hamming Weight in Boyar-Peralta Sbox (Note the Y-scale Range) 102

6.4 Partial Structure of the PPRM1 S-box Design 104

6.5 Partial Structure of Boyar-Peralta S-box Design 104

6.6 Timing Delays for FSA Resistant PPRM1 Design 108

6.7 Timing Delays for FSA Resistant AES Design for one Round 109

6.8 The architecture of the Trigger and Measurement Circuit for on-chip time measurements of S-box operations 111

7.1 Requirements of a successful fault attack 114

7.2 2-bit Ripple Carry Adder (RCA) 116

7.3 Observability Analysis for a Full Adder Circuit 123

7.4 Experimental Verification of Fault Generation Probability for PPRM1 Sbox 127

List of Tables

2.1 Fault Models Assumed by an Adversary 11

2.2 Fault Injection Mechanisms and Associated Fault Effects 13

4.1 Symbols of DFIA Attack Procedure 37

4.2 Key Hypothesis Table 40

4.3 Required Number of Physical Fault Intensity Levels and Glitched Clock Cycles for DFIA Attack on PRESENT and LED with 100ps Fault Injection Resolution 68

5.1 Cost of Fault Attacks for Noisy and Ideal Fault Injection Conditions 93

6.1 Data Dependency of Fault Sensitivity in AND, OR and XOR Gates 99

7.1 Table of Notations 119

7.2 Timing Characterization on $FA1$ for RCA example 121

7.3 Observability Computation Rules for Different Gates 122

7.4 $\frac{P_{obs}(Piret)}{P_{obs}(Tunstall)}$ for each output bit of SoV 131

7.5 Comparison of TVVF for Two Fault Attacks on Two Implementations . . . 131

Chapter 1

Introduction

The use of electronic systems has a rapid growth in day to day life. These devices have many applications from hand held devices such as mobile phones to critical applications such as aviation control and space craft telemetry. Due to the high utilization of these devices in human life, their security and privacy encounters new challenges.

Nowadays, cryptography is at the core of every application and secures almost all aspects of them. The basic principle of the asymmetric cryptography is a one way function that is easy to compute, but difficult to invert. Computing a message with this function is called encryption and inverting the encrypted message is decryption. In 1977, the first public key cryptosystem has been proposed, the so called, RSA.

Cryptanalysis is the study of analyzing the cryptosystems in order to find hidden or secret aspects of the system. There have been many proposed methods that analyze cryptosystems

mathematically such as linear and differential cryptanalysis. However, the security of a cryptosystem has been shown to be dependent to both the strength of the cryptographic algorithm and its physical implementation.

Secure cryptographic circuits are subject to a wide variety of cryptanalytic techniques, at the level of the algorithm as well as at the level of the implementation. Fault analysis is a class of implementation-oriented attacks. They analyze the response of a circuit to a fault injection, with the objective of accurately estimating an internally stored secret such as a key or a secret variable. Known since over a decade, fault analysis has grown into an advanced and refined cryptanalytic technique that handles public-key as well as secret-key cryptographic implementations [4, 5]. Faults can be obtained by pushing the circuit outside of its nominal operating conditions. Some common techniques include overclocking it, voltage-starving it [6], heating it up [7], creating spurious charges using optical means [8], or causing eddy currents through electromagnetic induction. Fault injection techniques are only limited by the creativity of the adversary.

A generic solution against faults is to use redundancy, such as by replicating the hardware implementation, by repeating computations, or by applying data error-coding techniques. The idea of redundancy is to tolerate sporadic faults by ensuring that at least part of the circuit obtains a correct result. The advantage of fault tolerant design is that it can handle (within some limits) any fault regardless of the fault location and fault timing in the circuit. However, fault tolerant design using redundancy is expensive. Spatial redundancy multiplies the hardware cost, and time redundancy reduces the performance, each by a factor of several

times. Full fault tolerance is therefore only available to systems that can afford over-design. Despite the costly overhead, most of these fault tolerant solutions are still not applicable to the fault attack problem because in these designs, the fault is assumed to be random and sporadic.

In fault attacks, faults are injected by an adversary rather than by nature. The adversary is intelligent and determined, rather than random and indifferent. The adversary also makes specific assumptions about the objectives of the fault attack, and about the algorithm being cryptanalyzed. Indeed, because of the widespread adoption of cryptographic standards, these assumptions are quite reasonable. This means that the objective of a fault attack is quite specific: the objective is to extract a secret key.

Another common assumption of the adversary is that many design details of the cryptographic implementation are known. Indeed, by using basic reverse engineering techniques, the adversary may learn the execution schedule of a cryptographic algorithm as it operates clock cycle by clock cycle, or the meaning of memory locations and registers used by the digital circuit. Knowledge of such design details is often helpful for a fault attack, and therefore the worst-case assumption is to assume that the adversary is fully knowledgeable about the implementation details of a cryptographic design.

Even though there has been an extensive research on fault attacks and countermeasures against these attacks, the following questions have not yet been answered in this area.

- The traditional fault attacks have many restrictions on the type of the fault that is

to be injected into cryptosystems for further analysis such as injecting single-bit fault, single byte fault, diagonal faults. Therefore, compared to the passive side channel analysis techniques, fault attacks have been considered a less powerful threat to the security of embedded systems. The question here is: if there exist novel models of fault attack that require less effort for the adversary?

- There have been many proposed countermeasures against fault injections with the objective of providing reliability and dependability for the embedded systems. These methods are mostly based on redundancy countermeasures. Are fault attacks able to break these countermeasures? If yes, how can we improve the resistance of these countermeasures against fault attacks?

- How secure is a system against fault attacks? The need for a design metric to assist designers with their security needs against fault attacks is increasing with the improvement of fault attacks over the years.

- How to systematically build a fault-attack resistant design? A fault attack has different intent compared to random fault injections. Therefore, the need for studying different aspects of a fault attacks from designer's point of view as well as attacker's point of view becomes one the concerns in nowadays embedded system design issues.

In this regard, this dissertation has three main contributions.

- We introduce Differential Fault Intensity Analysis, which combines the principles of Differential Power Analysis and fault injection. We observe that most faults are biased

- such as single-bit, two-bit, or three-bit errors in a byte - and that this property can reveal the secret key through a hypothesis test. Unlike Differential Fault Analysis, we do not require precise analysis of the fault propagation. Unlike Fault Sensitivity Analysis, we do not require a fault sensitivity profile for the device under attack. We demonstrate our method on an FPGA implementation of AES with a fault injection model. We find that with an average of 7 fault injections, we can reconstruct a full 128-bit AES key. We also show the DFIA attack on two lightweight block ciphers: PRESENT and LED. For each algorithm, our research analyzes the efficiency of DFIA on a round-serial implementation and on a nibble-serial implementation.We show that these algorithms and their implementation variants can be broken with 10 to 36 fault intensity levels, depending on the case. We also analyze the factors that affect the convergence of DFIA. We show that there is a trade-off between the number of required plaintexts, and the resolution of the fault-injection equipment. Thus, an adversary with lower-quality fault-injection equipment may still be as effective as an adversary with high-quality fault-injection equipment, simply by using additional encryptions. This confirms that DFIA is effective against a range of algorithms using a range of fault injection techniques.

- We also analyze the behavior of different implementations of AES S-box architectures against Fault Sensitivity Analysis (FSA), and propose a systematic countermeasure against this attack. In this regard, our countermeasure has two contributions. First, we study the behavior and structure of several S-box implementations, to understand

the causes behind the fault sensitivity and biased fault. We identify two factors: the timing of fault sensitive paths, and the number of logic levels of fault sensitive gates within the netlist. Next, we propose a systematic countermeasure against FSA. The countermeasure masks the effect of these factors by intelligent insertion of delay elements. We evaluate our methodology by means of an FPGA prototype with built-in timing-measurement. We show that FSA can be thwarted at low hardware overhead. Compared to earlier work, our method operates at the logic-level, is systematic, and can be easily generalized to bigger circuits.

- We introduce Timing Violation Vulnerability Factor (TVVF), which evaluates the vulnerability of a hardware structure against setup time violation attacks. The TVVF, a probabilistic metric computed on a circuit's netlist, is comprised of two factors: First, the probability of injecting a specific fault into the hardware structure, and second, the probability of propagating this fault to the output of the structure. TVVF aims at evaluating the security of designs against intentional faults caused by an adversary. In contrast, existing vulnerability metrics such as the Architecture Vulnerability Factor (AVF), evaluate the system reliability against random uncontrolled faults. To show the applicability of our metric, we compute the TVVF for two fault attacks on two AES netlists, which are generated for an FPGA.

The organization of this dissertation is as follows. Chapter 2 explains the basics of fault attacks and the steps that an adversary has to go through to build his attack. Chapter 3 explains a new type of fault model which is named biased fault model. Chapter 3 explains a

fault attack framework along with the new fault attack procedure named Differential Fault Intensity Analysis which is based on biased fault model. Chapter 4 provides a comparison between recently proposed biased-fault based attacks. Chapter 5 demonstrates the causes of biased behavior in circuits along with our proposed countermeasure against biased-based fault attacks specifically fault sensitivity analysis. Chapter 6 provides the designers with Timing Violation Vulnerability Factor that is a metric to measure the security of a hardware design against fault attacks. Finally, Chapter 7 concludes the dissertation.

Chapter 2

Background

In this chapter, we explain the fundamentals and requirements of fault attacks. Then, we provide a a hierarchy of actions that are required from the adversary for a successful fault attack.

2.1 Fault Attack

Electronic systems are subject to temporary and permanent faults caused by imperfections in the manufacturing process as well as by anomalies of the environment. Fault effects in electronics have been intensively studied in the context of system reliability as well as error resiliency. However, faults can also be used as a hacking tool. In a fault attack, an adversary injects an intentional fault in a circuit and analyzes the response of that circuit to the fault. The objective of a fault attack is to extract cryptographic key material, to

weaken cryptographic strength, or to disable the security. Unlike some other attacks, such as power-based or electromagnetic-based side-channel analysis, fault attacks do not require complex signal measurement equipment. The threat model of a fault attack assumes an adversary who can influence the physical environment of the electronic system - a condition that holds for a large class of embedded electronics such as smart cards, key fobs, access controls, embedded controllers, and so on. Fault attacks have been studied since the turn of the century, and today a great variety of methods are available to attack all forms of cryptography [9, 4, 10, 11].

In fault attacks, faults are injected by an adversary rather than by nature. The adversary is intelligent and determined, rather than random and indifferent. The adversary also makes specific assumptions about the objectives of the fault attack, and about the algorithm being cryptanalyzed. Indeed, because of the widespread adoption of cryptographic standards, these assumptions are quite reasonable. This means that the objective of a fault attack is quite specific: the objective is to extract a secret key. In the next two sections, we explain the requirements of fault attacks and the fault attack procedure.

2.2 Fundamentals of Fault Attacks

In a digital circuit, a fault is manifested through a temporary or permanent change in the correctness of computations. Faults in digital systems originate from a variety of causes, related to manufacturing issues as well as to environmental issues. Of primary interest to

fault attacks are intentional faults caused by an adversary, as opposed to faults that have a random, uncontrolled cause. A successful fault attack is composed of a fault measurement and the fault analysis process. The fault analysis process is based on the information leaked while building the fault model. The aim of the attacker is to be able to inject an intentional fault, using a series of techniques to manipulate the environmental conditions of a circuit, that results in the desired fault model. In this section, the emphasis is on the requirements for fault measurement. Two preliminary concepts to understand fault measurement, are fault model and fault injection. We will first explain these two, and then explain how they relate to fault measurement.

2.2.1 Fault Model

The fault characteristics resulting from a fault injection are commonly captured in a *fault model*, which is also the starting point of various cryptanalytic methods. A fault model expresses the important fault characteristics: the location of the fault within the circuit, the number of bits affected by the fault, and the fault effect on the bits (stuck-at, bit-flip, random, set/reset).

Table 2.1 lists four common fault effects: Chosen-bit Fault, Single-bit Fault, Byte Fault, and Random Fault. For example, in chosen bit fault model, the attacker must precisely select the location of the faulty bit and change its value to either 0/1.

Table 2.1: Fault Models Assumed by an Adversary

	Fault Location	Number of Bits	Fault Effect
Chosen Bit Fault	Precise	1	set/reset
Single Bit Fault	Loose	1	stuck-at
Byte Fault	Loose	8	stuck-at
Random Fault	Loose	any	random

2.2.2 Fault Injection Techniques

As mentioned, the objective of the attacker is to build the fault model for a successful post-processing of the information. There are several fault injection tools and techniques for building the fault model. The following are six possible mechanisms of fault injection.

- **Clock Glitches** are used to shorten the clock period of a digital circuit during selected clock cycles [12, 13]. If the instantaneous clock period decreases below the critical path of the circuit, then a faulty value will be captured in the memory or state of the circuit. An adversary can inject a clock glitch by controlling the clock line of the digital circuit, triggering a fault in the critical path of the circuit. If the adversary knows the circuit structure, he or she will be able to predict location of the circuit faults. Glitch injection is one of the methods that can be obtained in many different ways(EM pulse, voltage glitch, clock glitch), and therefore it can be considered as a broad threat to secure circuits.

- **Voltage Starving** can be used to artificially lengthen the critical path of a circuit, to a point where it extends beyond the clock period [14]. This method is similar to injection of clock glitches, but it does not offer the same precise control of fault timing.

- **Voltage Spikes** cause an immediate change in the logic threshold levels of the circuit [9]. This changes the logic value held on a bus. Voltage spikes can be used, for example, to mask an instruction read from memory while it's moving over the bus. Similar to clock glitches, voltage spikes have a global affect and affect the entire circuit.

- **Electromagnetic Pulses** cause Eddy currents in a chip, leading to erroneous switching and isolated bit faults [15]. By using special probes, EM pulses can be targeted at specific locations of the chip.

- **Laser and Light Pulses** cause transistors on a chip to switch with photo-electric effects [16]. Through focusing of the light, a very small area of the circuit can be targeted, enabling precise control over the location of the fault injection.

- **Hardware Trojans** can be a source of faults as well. This method requires that the adversary has access to the circuit design flow, and that the design is directly modified with suitable trigger/fault circuitry. For example, recent research reports on an FPGA with a backdoor circuit which disables the readback protection of the design [17].

The fault injection mechanism determines the timing of the fault, the duration of the fault (transient or permanent), and the fault intensity (weak/strong).Together, these characteristics enable the adversary to select a specific fault model, which is needed as the starting

Table 2.2: Fault Injection Mechanisms and Associated Fault Effects

Injection	Fault Characteristic			Fault Model*				Invasiveness
	Intensity	Timing	Duration	Chosen Bit	Single Bit	Byte	Random	
Glitches	Variable	Precise	Transient			•	•	Non-invasive
Starving	Variable	Loose	Transient			•	•	Non-invasive
Spikes	Fixed	Precise	Transient				•	Non-invasive
EM Pulse	Variable	Precise	Transient		•	•	•	Non-invasive
Laser Pulse	Fixed	Precise	Transient	•	•	•	•	Semi-invasive
Trojans	Fixed	Precise	Permanent	•	•	•	•	Invasive

* • means the Fault Model can be generated with this fault injection method

point of cryptanalysis by fault injection. The method of fault injection also influences the difficulty of performing it. Depending on the level of tampering required with the actual circuit, one distinguishes non-invasive, semi-invasive [18] and invasive attacks. Table 2.2 illustrates the relation between the aforementioned six possible fault injection mechanisms, along with the fault models resulting from their use.

2.3 Fault Attack Procedure

Any fault attack consists of two phases: a *fault measurement* phase, and a *fault analysis* phase.

Based on the adversary's access to the target device and the information required to attack a specific block cipher, the adversary aims for a fault model. Therefore, choosing the fault

model is a part of the fault analysis process. Then, in the measurement process, the adversary goes through the four steps to build the fault model using actual measurements. In this section, we first explain the fault measurement phase and then provide an example of fault analysis.

2.3.1 Fault Measurement

A fault model and a fault injection mechanism to trigger the fault model are two essential ingredients of a fault attack. But to apply them in a successful fault attack, we need to consider a larger scope. Figure 2.1 shows that a successful fault attack consists of two steps, fault measurement and fault analysis. The fault injection is part of the fault measurement phase, while the fault model is a building block in fault analysis.

Indeed, Figure 2.1 shows both the requirements of a successful fault attack and the principles of the fault-attack resistant design. From the adversary's point of view, each step of the pyramid should be followed in order. An adversary first needs to choose a fault model and fault analysis technique based on the target cryptosystem, Then, he needs to obtain exploitable faults by following the steps of fault measurement, from fault injection access to fault observation.

From the designer's side, the steps of this pyramid should be considered while designing a fault-attack resistant device. For each step, the designer should evaluate the costs and benefits of securing the design against this step. Using the evaluation results, the designer

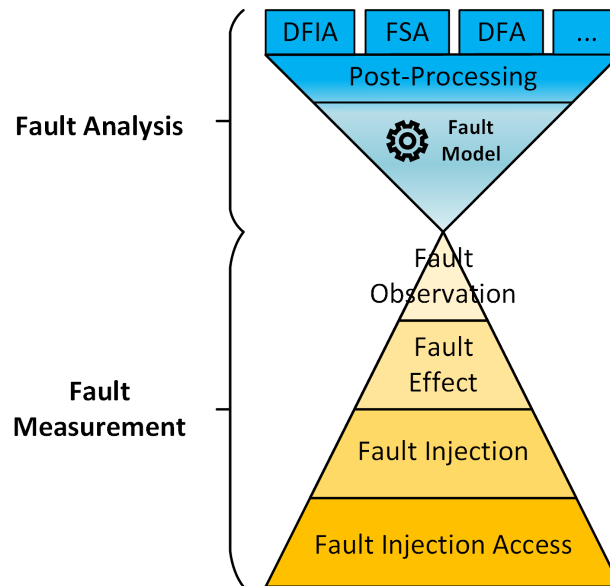


Figure 2.1: A Fault Attack Requires Fault Measurement and Fault Analysis

is able to make design decisions to prevent the adversary from building the required fault model. Next, we explain the steps of fault measurement and demonstrate them using a case study.

2.3.2 Fault Measurement Steps

The reality of fault measurement is more complicated than injecting a fault. First of all, the adversary needs to be able to physically inject a fault. The fault injection also needs to have the desired effect, and result in an exploitable fault, that results in the required fault model. Finally, the exploitable fault needs to be observable.

Following is a more comprehensive definition of these four levels.

- *Fault Injection Access*: The first and foremost step of the fault measurement is getting physical access to the device under test (DUT). For example, an adversary needs to control external clock and supply voltage ports of DUT for clock and voltage glitching attacks, respectively [13]. Similarly, the adversary must have physical access to chip surface for laser and electromagnetic pulse-based fault attacks [19]. In addition, an adversary may also need to control data inputs and outputs of DUT. The amount of physical access needed for each attack is different.

- *Actual Fault Injection*: The second step of the fault measurement is disturbing the operation of DUT by applying a physical stress on it. The applied physical stress pushes DUT out of its normal operating conditions and causes faulty operation. Based on the chosen fault injection method, the adversary can control the timing, location, and intensity of the applied physical stress. Each value of these three parameters affects DUT differently and causes different faults in DUT operation. Therefore, the adversary needs to carefully set these parameters to create an exploitable fault in DUT operation. In clock glitching, for example, the adversary causes setup time violation by temporarily applying shorter clock cycles. The adversary can control the timing and length (i.e, intensity) of the applied shorter clock cycle. However, there is no control on the location of the applied physical stress (i.e, a shorter clock cycle) in this case because clock is a global signal for DUT.

- *Fault Effect*: The third step is creating a fault effect on DUT operation as a consequence of the applied physical stress. The fault effect can be defined as the logical (or digital)

effect of the applied physical stress on DUT operation. For example, an applied clock glitch might create 2-bit faults at the fault injection point. Similarly, a laser pulse might affect 1-bit of DUT. On the other hand, it is also possible to not create any fault effect even though a physical stress is applied on DUT. The adversary has a limited and indirect control on the fault effect through controlling the physical stress. The fault effect depends on various factors such as circuit implementation, used fault injection method, applied physical stress, etc. The adversary may need to apply several physical stresses with different parameters to create the desired fault effect on DUT operation [2][3].

- *Fault Observation*:The final step towards the fault measurement is to observe the effects of the fault injection in the output of the block cipher or the algorithm under the attack. The authors of [20], show that there are methods that can compute the probability of success of a fault injection attempt in this phase using observability analysis. This method basically computes the probability of observability from the point of fault injection to the output of the cipher. In this work, we use a probability-based observability analysis method for the probability of propagating an exploitable fault to the output. Observability analysis, which is widely used in VLSI test area, reflects the difficulty of propagating the value of a signal to primary outputs [21].

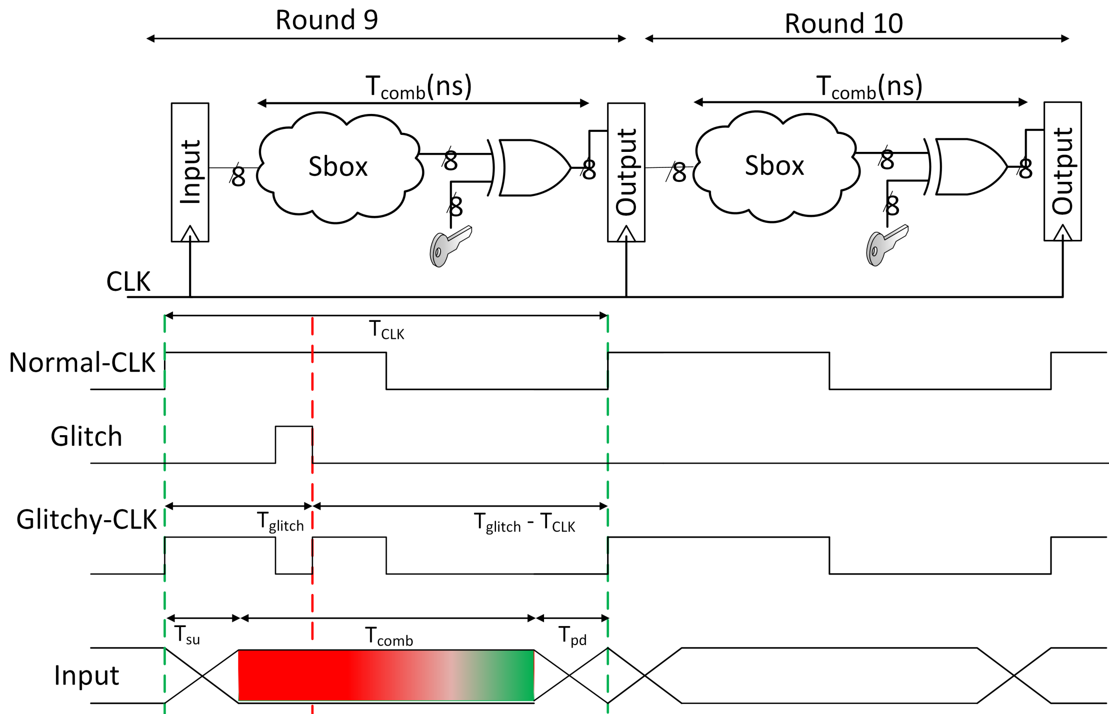


Figure 2.2: Sample Combinational Logic

2.3.3 Fault Measurement Case Study

In this section, we apply the ideas of fault measurement to a specific example. Figure 2.2 shows a simple combinational logic of two rounds of a hypothetical block cipher. Each round is composed of an SBOX module and an XOR gate. SBOX is a substitution module that obscures the relationship between the secret key and the output. To explain the fault model and fault injection steps, we assume that the intentional fault model required for an attack is to inject a random fault into the output of the combinational logic in round 9 shown in Figure 2.2. The injected fault must then be propagated through round 10 and be observable by the adversary in the output of round 10.

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:

$$T_{clk} \geq T_{pd} + T_{su} \tag{2.1}$$

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.

In this case, 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 [22]. In the following paragraphs, we explain fault measurement process for fault injection using setup time violation.

- *Fault Injection Access*: 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. The attacker must have access to the clock signal that is driving this circuit.

- *Actual Fault Injection*: An adversary can cause setup time violation via clock glitches. Figure 2.2 shows the effect of a glitch on the clock signal. As shown, the glitch signal is

XORed with the normal clock. During round 9, a clock glitch will temporarily shorten the clock cycle period from T_{clk} to T_{glitch}, thereby causing timing violation of the digital logic.

- *Fault Effect*: 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. As shown in Figure 2.2, the computation of the data in the combinational logic is not yet finished and it is captured as output due to the glitchy clock. Therefore, the captured value can be faulty.

- *Fault Observation*: To observe the effects of fault injection, the output of round 10 must be different from the correct value. Therefore, the effect of injected fault into the output of round 9 must be propagated through round 10 without being masked. If the value of output 9 is different from its correct value and is not masked by the round 10 operations, the fault measurement process is successful.

2.3.4 Fault Analysis: Differential Fault Attack

In this section, we explain the fault attack process with three different fault models. All of the example attacks are on the Advanced Encryption Standard (AES) algorithm. The details of this algorithm are explained in the following section.

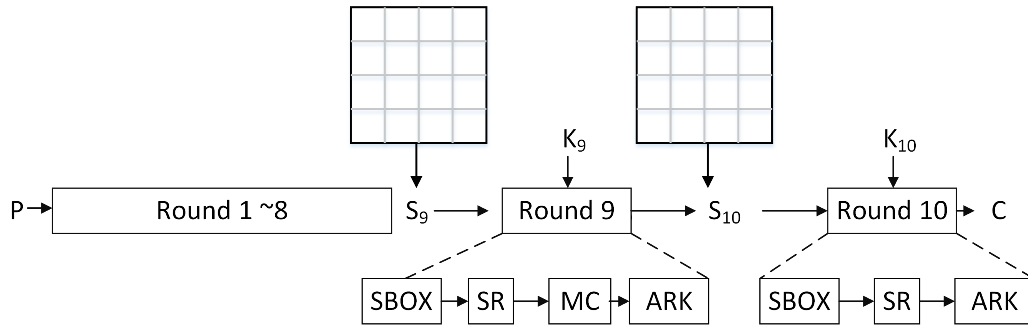


Figure 2.3: AES Structure

Advanced Encryption Standard

The AES algorithm consists of 10 rounds. The first 9 rounds have 4 main operations, SBOX, ShiftRows(SR), MixColumns(MC) and AddRoundKey(ARK). Round 10 omits the MixColumn operation. Figure 2.3 shows the structure of the AES algorithm. In this figure, P is the applied plaintext to the AES algorithm, S_{10} is the intermediate state variable for round 10. K_{10} is the key for round 10 and C represents the ciphertext. The faulty value of the variable x is shown by x'.

DFA is one of the most studied types of attack on cryptographic systems such as RSA [23], DES [24] and AES. DFA assumes that the attacker is in possession of the device and is able to obtain two faulty and fault free ciphertexts for the same plaintext. DFA also assumes that the attacker is aware of some characteristics of the injected fault. There are many proposed types of DFA attack on the AES algorithm [25],[26],[27]. These attacks are based on different fault models and choose various methods of injection techniques based on the fault model. In this section, we will explain the steps of a simple electromagnetic pulse-based DFA attack

on AES, which is proposed by Dehbaoui et al. [28].

Fault Model: This attack adopts Piret's fault model [29]. This fault model requires an adversary to induce one byte fault in AES state between the start of round 9 and the MixColumns operation.

Fault Measurement: The DUT for the attack is a RISC microcontroller running AES algorithm. In this attack, the adversary injects faults by means of transient electromagnetic (EM) pulses. A small magnetic coil is used apply EM pulses without any physical contact to DUT. The adversary can control the timing, energy, and position of the applied EM pulses. By using different combinations of these three parameters, the adversary can affect only one byte of the computation and can select the affected byte. As a result, an adversary can induce the exploitable faults using this setup.

Fault Analysis: Due to the MixColumns in round 9, one faulty byte in the beginning of round 9 will cause 4 faulty bytes in the ciphertext. Therefore, we can find 4 bytes of the key of round 9. Assuming that the attacker only injects fault into one byte, the C and C' differ in four bytes. There are 255×4 possible values for these four bytes which is saved in a list D. For each key guess, the adversary should compute the value of these four bytes using inverse equations of AES operations. The key is potentially a correct candidate if the computed value is in the list D. The adversary should continue injecting fault in the same location until only 1 key remains as the candidate.

Chapter 3

Biased Fault Models

In this section, we explain the challenges and the requirements of the traditional fault attacks. These requirements are basically based on the assumptions on the fault model in the traditional fault attacks. Based on our studies, we propose a new fault attack that is founded on a fault model which is the realistic behavior of a circuit under fault injection.

3.1 Challenges of Traditional Fault Attacks

Secure cryptographic circuits are subject to a wide variety of cryptanalytic techniques, at the level of the algorithm as well as at the level of the implementation. Fault analysis is a class of implementation-oriented attacks. They analyze the response of a circuit to a fault injection, with the objective of accurately estimating an internally stored secret such as a key or a secret variable.

On the other hand, fault attacks face a recurring challenge. The adversary needs to ensure that the actual physical manifestation of the fault corresponds to the fault model assumption made during fault analysis. Such assumptions include, for example, the location of the fault in the circuit, the precise time at which a fault must occur, and the specific value of a faulty variable. These conditions are summarized in a *fault model*, the set of properties that describe a given fault. Some of the well known fault types assumed by fault attacks are random-byte errors, bit-flip errors, or stuck-at errors. The attacks further assume a specified time precision that can range from a complete encryption period, to one cryptographic round, down to a precise clock cycle.

The other challenge to the cryptographic engineer is to ensure that the available fault injection techniques for the circuit under consideration, will provide the fault model required for the selected fault analysis. That is a challenge, for various reasons. First, the resolution of fault injection techniques varies greatly. Some fault injection techniques only enable time control, but are imprecise in terms of location. Glitch injection and electromagnetic-pulse injection fall in this category. Some fault injection techniques only have a global effect. Temperature and voltage fall in this category. On the other hand, precise fault injection may be too expensive or too complicated. For example, the use of a laser to trigger setup effects in a selected register, requires partial disassembly of a chip package.

The efficiency of a fault attack is inversely proportional to the number of fault injections required to learn an internal secret. In well-known fault attacks such as differential fault analysis, a more precise fault model typically requires fewer faults [30]. Hence, a more

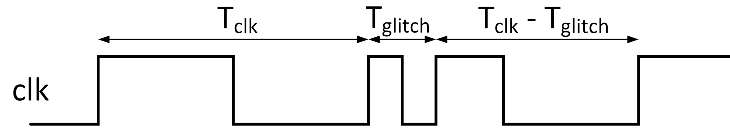


Figure 3.1: The effect of the glitch injection on the clock signal

precise fault model is therefore a desirable objective in fault analysis. On the other hand, due to the above mentioned reasons, guaranteeing a precise fault model is very difficult and costly.

3.2 Clock Glitch Injection Mechanisms

In this section, we first explain the mechanism of clock glitch injection, and then, describe the causes of fault bias.

3.2.1 Setup Time Violation

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 [22]. In the following paragraphs, we explain setup time constraints of a circuit and their use as the fault injection means.

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:

As mentioned in Chapter 2, Equation 2.1 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.

An adversary can cause setup time violation via clock glitches. Figure 3.1 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.

3.2.2 Causes of Fault Bias

Recently, a series of fault analysis techniques have been introduced that are based on *fault bias*. In this context, fault bias is the proportion of a circuit that experiences a fault under a given fault injection. Attacks based on fault bias can use relaxed fault models, compared to classic fault attacks.

The fault behavior of a circuit under clock glitch injection is determined by its *path delay*

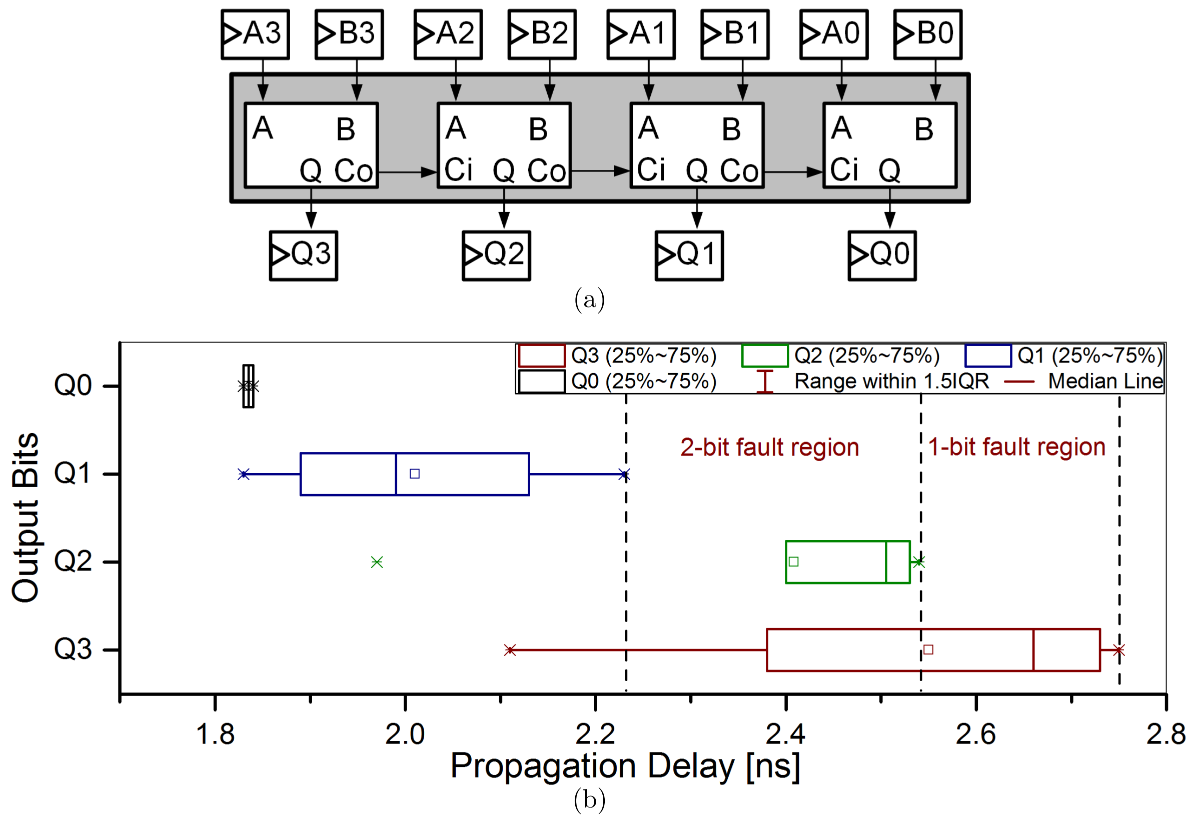


Figure 3.2: **(a)** A block diagram of a 4-bit ripple-carry adder implemented on an FPGA. **(b)** The distribution of path delays within fan-in cone of each output bit. Using the non-uniform path delay distribution, we can create biased faults (e.g, 1-bit, 2-bit, etc.)

distribution and the applied *fault intensity*. For clock glitching, we define the *fault intensity (FI)* as the the inverse of glitch period, T_{glitch} (Fig. 3.1), and quantify it with Equation 3.1. The following paragraphs explain how these two factors can be combined to inject biased faults.

$$FI = \frac{1}{T_{glitch}} \tag{3.1}$$

Figure 3.2(a) shows the block diagram of a 4-bit ripple-carry adder, which computes sum (Q) of two input numbers (A, B). Figure 3.2(b) shows the (static) path delay distribution of the adder in the form of box-whisker plots. Each box-whisker plot in Figure 3.2(b) shows the delay distribution of paths within the fan-in cone of a different output bit. We extracted the path delays from a post-place-and-route netlist generated for a Xilinx Spartan 3E FPGA. As it is seen, the path delay distribution is non-uniform. For example, more than 50% of the path delays within the fan-in cone of bit $Q3$ are greater than all of the path delays within the fan-in cone of bit $Q2$. Similarly, the path delays within the fan-in cone of $Q0$ are the smallest ones. This observation promises a biased (i.e, non-uniform) fault behavior. For example, it is possible to inject a fault that affects only a few bits of a targeted variable. This can be achieved by applying a fault intensity that violates only a few paths. If the applied fault intensity is greater than $0.364GHz$ (i.e, $T_{glitch} > 2.75ns$), no faults can be injected into the ripple-carry adder. However, single-bit faults can be injected on the bit Q_3 when the fault intensity is between $0.364GHz$ and $0.392GHz$ (i.e, $2.55ns < T_{glitch} < 2.75ns$). Additional faults can be induced in the other bits only if the fault intensity value is increased further.

As a consequence, the non-uniform path delay distribution enables an adversary to obtain a biased fault behavior in proportion to the fault intensity. The next section provides a definition for the fault bias, which allows us to quantify it.

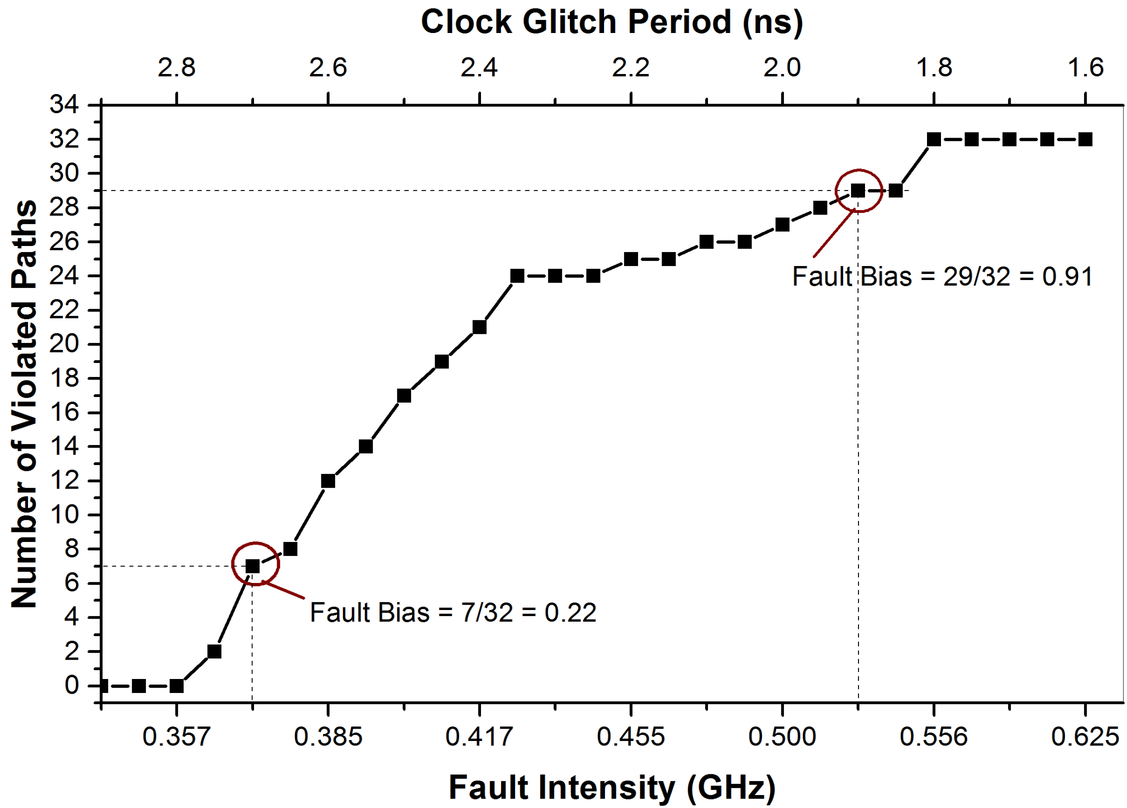


Figure 3.3: Number of violated paths with respect to the applied fault intensity and the corresponding glitch period for the ripple-carry adder.

3.2.3 Quantifying Fault Bias

The fault bias is a property of the circuit architecture, which expose the potential of a circuit to experience a setup time violation at a given fault intensity. Therefore, we can define the *fault bias (FB)* as the proportion of the violated paths for a given fault intensity. This definition enables us to quantify the fault bias as a number between 0 and 1 via Equation 3.2.

$$FB(f) = \frac{Number\ of\ violated\ paths}{Number\ of\ all\ paths}\bigg|_{FI=f} \tag{3.2}$$

Figure 3.3 illustrates this concept for our ripple-carry adder example. It shows the number of violated paths with respect to the fault intensity and the corresponding clock glitch period. The bottom horizontal axis shows the fault intensity values and the top horizontal axis shows the corresponding glitch period values. As it is seen, the number of violated paths increases with the fault intensity. As an example, we will explain two fault bias values for two different fault intensities. The first fault intensity is $0.370 GHz$ and the corresponding glitch period is $2.70 ns$. The fault bias for this fault intensity value is 0.22. This means that only 22% of the all paths can contribute the fault behavior. As it is shown in the ripple-carry example, this fault intensity can induce only 1-bit faults (Fig. 3.2(b)). The second fault intensity and the corresponding glitch period are $0.526 GHz$ and $1.90 ns$, respectively. At this fault intensity, the fault bias is 0.91. In this case, only the bit $Q0$ of the adder computes the correct result. Depending on the attack strategy, both of these cases can be exploited to retrieve the key.

Equation 3.2 reveals three important properties of fault bias. First, the fault bias is a property of circuit architecture. Therefore, an adversary can accurately model the fault behavior in terms of circuit architecture. Second, an adversary can control the severity of fault effects on a circuit by controlling the fault intensity. Higher fault intensity values induce more severe fault effects. Thus, each point in Figure 3.3 provides additional information that can be used as a source of leakage. Third, cryptographic hardware designers can evaluate the vulnerability of their designs to setup time time violation, and develop more efficient countermeasures. Next, we will demonstrate other parameters that affect the fault bias.

3.2.4 Effects of Operating Conditions and Data on Fault Bias

During an attack, the adversary can influence the dynamic path delay distribution through *the input data* and *the operating conditions.* This will affect the fault bias. In the following paragraphs, we will demonstrate their effects on the path delays.

Effects of Operating Conditions

An adversary can influence the path delays of a circuit by varying the operating conditions such as the supply voltage and the operating temperature. In Figure 3.4, we provide an example to illustrate the effects of varying the supply voltage and the operating temperature on the path delays of the ripple carry adder. We investigate the path delay distribution within the fan-in cone of bit Q_3 with four different (*supply voltage, operating temperature*) combinations. We provide a separate box-whisker plot for each case. We obtained these data using Xilinx's static timing analyzer tool.

The top two box-whisker plots of Figure 3.4 show the effect of increasing the operating temperature from $85C$ to $100C$, while applying a constant supply voltage of $1.14V$. The bottom two box-whisker plots show the same effect for the supply voltage of 1.32. As it is seen, the path delays increase with the increasing temperature.

The *Case 1* and *Case 3* shows the effect of decreasing the supply voltage from $1.32V$ to $1.14V$ for a temperature of $85C$. The *Case 2* and *Case 4* illustrate the same effect for a temperature of $100C$. As it is shown, the path delays increases with the decreasing supply

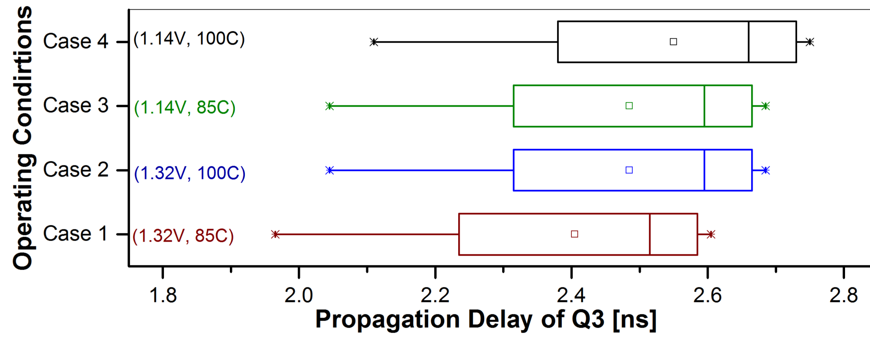


Figure 3.4: The effect of varying the supply voltage and the operating temparature on the path delays. Increasing the temperature and decreasing the supply voltage increase the path delays.

voltage. These behaviors are compatible with the experimental results provided by Zussa et al. [31].

Effects of the Processed Data

An adversary can also influence the path delays via processed data since path delays are data-dependent.

In Figure 3.5, we provide post-place-and-route simulation results for our ripple-carry adder example to illustrate the data-dependency of its path delays. In the simulation, we first initialized the adder ouputs to *logic-1*. Then, we applied two different input sets and observed the timing of the output signals during their transition from *logic-1* to *logic-0*. As it is seen, path delays have a distribution of $T_{Q3} > T_{Q2} > T_{Q1} > T_{Q0}$ for the first input set (Fig. 3.5(a)). The output bit $Q3$ settles down later than the other bits because of the ripple effect. On

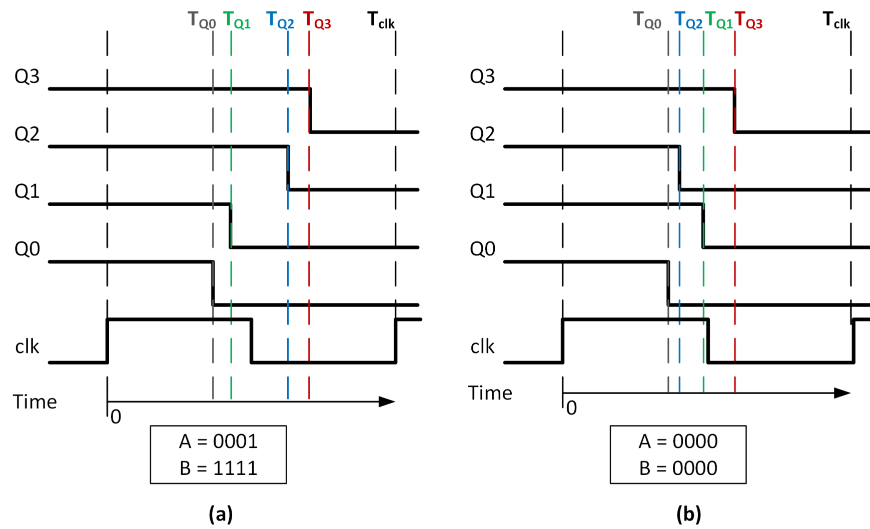


Figure 3.5: Illustration of data-dependency of paths delays on the ripple-carry adder: **(a)** $T_{Q3} > T_{Q2} > T_{Q1} > T_{Q0}$. **(b)** $T_{Q3} > T_{Q1} > T_{Q2} > T_{Q0}$.

the other hand, the path delay distribution is $T_{Q3} > T_{Q1} > T_{Q2} > T_{Q0}$ for the second input set (Fig. 3.5(b)). As there is no carry propagation in this case, the bit $Q1$ can settle down before the bit $Q2$. This shows that modifying the processed data changes the distribution of the path delays.

Chapter 4

Differential Fault Intensity Analysis (DFIA)

In the previous chapter, we discussed a systematic description of fault bias, its causes and effects on the circuit behavior. In this chapter, we introduce a new attack based on the concept of fault bias.

4.1 Fault Attack Framework

We first describe a generic framework for a systematic comparison of the fault attacks. Following are the steps of a fault attack. These steps are shown in Figure 4.1 as well.

- **Step 1: Measurement**: In this step, the adversary applies several plaintexts and all

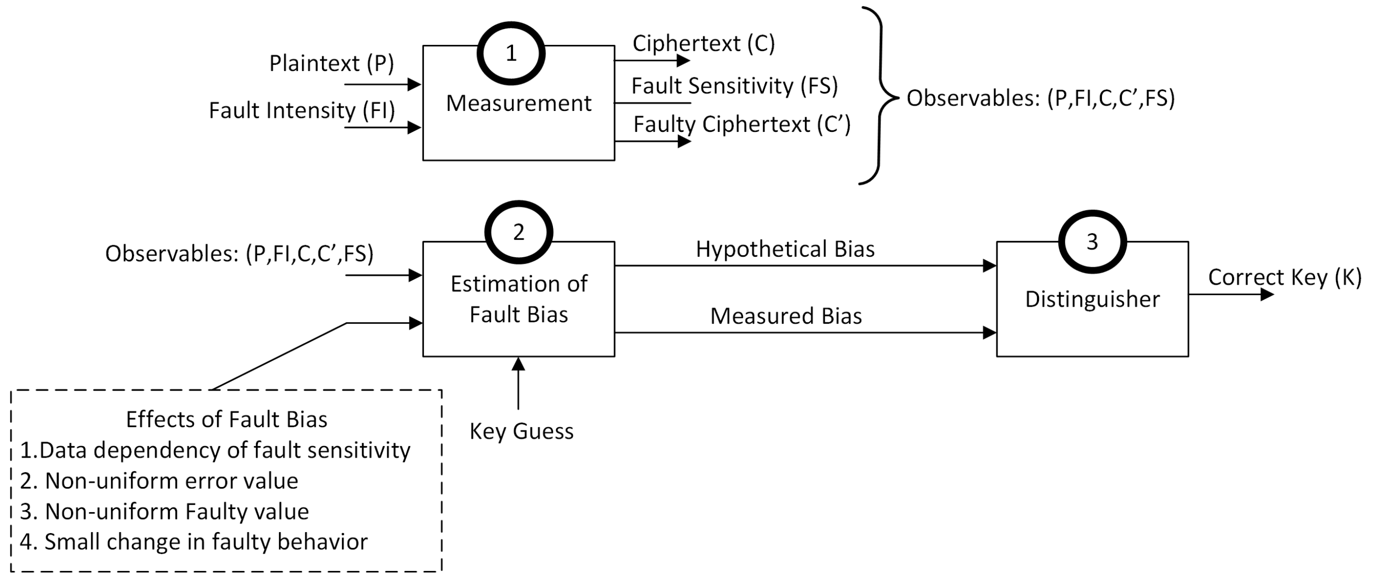


Figure 4.1: Generic Fault Attack Framework

possible fault intensities to inject biased fault into the block cipher. Then, he collects the observables, namely, plaintexts, faulty ciphertexts, correct ciphertexts and fault intensities for further analysis. The collected observables contain fault bias information of the circuit.

- **Step 2: Estimating the Effects of Fault Bias on Secret Intermediate Variable**: The target of biased fault injection is a secret intermediate variable. Therefore, an adversary should first invert the observables back to the intermediate variables using key guesses. Then, the adversary estimates the effect of fault bias on hypothesized intermediate variable. Since the effects of fault bias are different, each attack strategy requires a different estimation function on the intermediate variable.

- **Step 3: Distinguisher**: Only for the correct key guess, the estimated fault bias

values in Step 2 correspond to the observed fault bias in Step 1. In this step, we distinguish the correct key guess from the wrong key guesses. For this purpose, the distinguisher first assigns a number for the strength of the correlation between the collected observables (Step 1) and the fault bias estimations (Step 2). Then, it selects the key guess which corresponds to the maximum strength.

4.2 Differential Fault Intensity Analysis

The problem definition is as follows: Given a cryptographic algorithm, the fault injection tool and enough pairs of faulty ciphertexts corresponding to the same input, we want to find the round key. We make the following assumptions:

- The adversary knows what cryptographic algorithm is executing.
- The adversary is able to inject a fault in an intermediate variable S that produces a key-dependent observable output. This is similar to the way in which DPA selects intermediate variables.
- The fault can be injected with varying intensity, with a limited change to the intermediate variable.

Table 4.1 defines the notations of the DFIA attack procedure algorithm. DFIA uses three steps. Algorithm 1 explains the DFIA attack procedure. This algorithm consists of the following steps.

Table 4.1: Symbols of DFIA Attack Procedure

P	Plaintext
Q	Total number of injected faults with different intensities
q	A specific fault injection
C	Correct ciphertext
C'_q	Faulty ciphertext under fault q
S	Correct state
k	Key hypothesis
$S'_{k,q}$	Faulty state under hypothesis $K = k$, fault q, $S'_k, q = f(C'_q, k)$

Step 1: The attacker induces a fault into S while running the cryptographic algorithm with input P. The fault will change the value of S to a faulty value. The output of the algorithm will be C'_q.

The attacker is in possession of the values of C'_q. Under a key hypothesis k, he can compute $S'_{k,q}$ as a function of C'_q and k. The attacker will generate $S'_{k,q}$ for every possible key hypothesis.

Step 2: The attacker computes Equation 4.1 for each key hypothesis k. This equation adds up the Hamming Distance between the value of $S'_{k,q}$ for current fault injection and any previous one. The Hamming Distance between $S^{'}_{k,n}$ and $S^{'}_{k,m}$ calculates the difference between the number of injected faults in fault injection n and fault injection m.

Algorithm 1 DFIA Attack Procedure

Assume *Cryptographic Algorithm, Fault Injection Tool*;

Result *Correct Key Guess*;

//**Step1** *Fault Injection*;

forall the *Fault q such that* $1 \leq q \leq Q$ **do**

  Obtain faulty ciphertext C'_q;

  forall the *Key Hypothesis* k **do**

    Obtain faulty state $S'_k, q = f(C'_q, k)$;

  end

end

//**Step2** *Post-process*;

forall the *Key Hypothesis* k **do**

  Calculate $\rho_k = \sum_{n=1}^{Q} \sum_{m=1}^{n-1} HD(S'_{k,n}, S'_{k,m})$;

end

//**Step3** *Key Hypothesis*;

$K = argmin \rho_k$;

$$\rho_k = \sum_{n=1}^{Q} \sum_{m=1}^{n-1} HD(S'_{k,n}, S'_{k,m}) \tag{4.1}$$

Step 3: The attacker evaluates the Hamming Distance among faulty state variables testing for the distance that is most likely under the assumed biased fault model. To get to this point, the attacker should look for the minimum of cumulative Hamming Distance (ρ_k) among all key hypotheses. The reason behind looking for minimum is that only under the correct key guess the number of injected faults will correspond to the fault intensity, otherwise the Hamming Distance will be a random or unpredictable number. If no conclusion can be made on the most likely key, the attacker will increase Q and repeat the process.

Hence, similar to DPA, the DFIA attack tests the most likely model corresponding to an observation (measurement). Adding more observations improves the hypothesis test.

4.3 DFIA on Symmetric Key Cryptographic Primitives

In this section, we explain the key retrieval process of the DFIA algorithm on three different symmetric key cryptographic algorithms, considering different implementations of these algorithms.

4.3.1 DFIA on AES

In this section, we will explain the DFIA attack on the AES algorithm. AES algorithm is described in Section 2.3.4 (2.3). We use the same notations as before, with the extension

Table 4.2: Key Hypothesis Table

		Key Hypothesis K			
		k_0	k_1	...	k_n
Fault Injection	Faulty Ciphertexts	Apply Equation 4.3 to get S'_q			
1	C'_1	$S'_{0,1}$	$S'_{1,1}$	...	$S'_{n,1}$
2	C'_2	$S'_{0,2}$	$S'_{1,2}$	...	$S'_{n,2}$
...	...	...	...	...	...
Q	C'_Q	$S'_{0,Q}$	$S'_{1,Q}$	...	$S'_{n,Q}$
Apply Equation 4.1 to get ρ_k		ρ_0	ρ_1	...	ρ_n

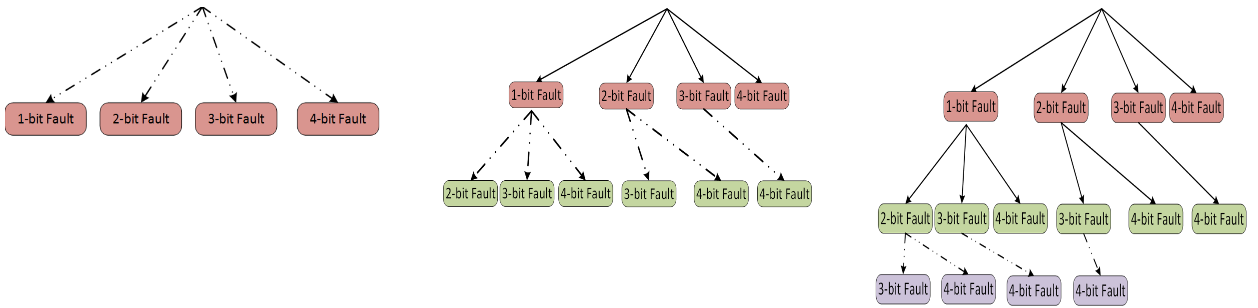
that S represents the AES state at round 10, and K represents the round key for round 10. If a fault q is induced in S, it will only corrupt a single byte of C since the last AES round does not perform MixColumns diffusion. The adversary then collects multiple faulty ciphertexts C'_1, C'_2, ..., C'_Q under the same plaintext intput P. We know that

$$C'_q = k \oplus ShiftRows(SubBytes(S_q{}')) \tag{4.2}$$

Therefore, the faulty state can be computed as

$$S_q{}' = SubBytesInverse(ShiftRowsInverse(k \oplus C'_q)) \tag{4.3}$$

We arrange the observations in a table such as Table 4.2, which we call the key hypothesis



(a) First Level of Fault Injection (b) Second Level of Fault Injection (c) Third Level of Fault Injection

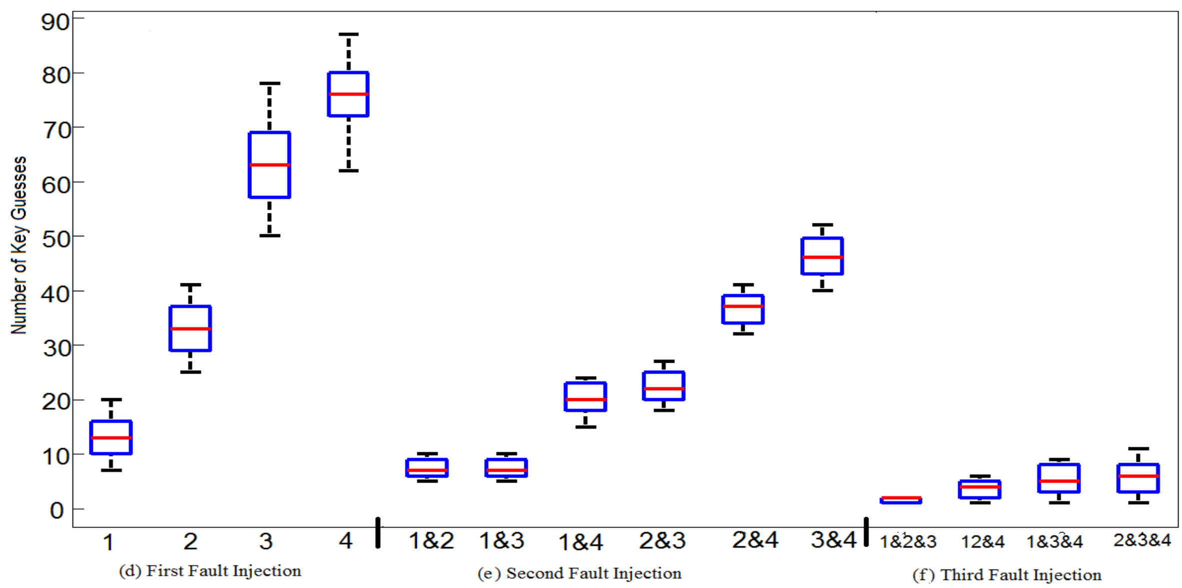


Figure 4.2: Effect of Choosing Fault Set on Key Recovery Convergence

table. Now, only one of the 256 key guesses is true. Because of the biased fault model, the true faulty state bytes must be close to each other in terms of Hamming Distance. Hence, under a given key hypothesis, we test the distance between faulty state bytes by calculating ρ_k. Only for a correct key guess will a minimum be found. As before, we may need multiple fault injections to uniquely pinpoint the most likely roundkey.

4.3.2 Methodology for Selecting Fault Set

The previous section demonstrated how a set of biased faults can be used to construct a hypothesis test leading to the correct key. In this section, we elaborate how to construct the set of biased faults such that the hypothesis test shows the quickest convergence. From a practical point of view, the adversary is interested in using as few faults as possible, since every fault injection costs time and effort.

To build an understanding of what makes a good set of biased faults, we performed the following experiment. We wrote a simulator for the AES algorithm that supports the injection of biased faults in the last round. The simulator allows the injection of single-bit, two-bit, three-bit and four-bit errors in a state byte. For example, injection of a two-bit biased fault in a state byte means that the simulator randomly selects two bits from the state byte. Then, it flips the selected bits to random values of 0 or 1. Hence, two-bit fault injection means that at most two bits in a byte have faulty values. Observe that this fault model only gives us control over the *number* of bits that may be affected, not over their location or their final value.

We argue that the single-bit, two-bit, three-bit and four-bit biased fault models can be used to simulate variation of fault intensity. At low fault intensity, a single-bit fault is more likely, while at higher intensities, multi-bit faults are more likely. This effect has been observed by other authors as well [6], [32] , and we also confirmed it through practical experiments (See section IV.B).

Using the simulator, we can now explore different strategies for fault injection campaigns. The most simple set of biased faults is a set with one single fault at a given intensity level (single-bit, two-bit, three-bit or four-bit). After injection of a fault, the simulator computes the number of possible key bytes under the given fault model. For example, after injection of a two-bit fault, the simulator finds the number of key candidates that, according to Equation 4.3, predicts a faulty state with a Hamming Distance of 1, or 2 from the original state. We repeat this for 256 different plaintext inputs, each time finding the number of key candidates.

Figure 4.2(top) shows different scenarios of injecting faults into the state byte. Each level of the tree shows different fault intensities at each trial. Figure 4.2(a) shows the possible fault intensities for the first fault injection trial. In the first trial, the attacker can inject either single-bit, two-bit, three-bit or four-bit faults into the state byte. Figure 4.2(d) shows how many key guesses could be the correct key after the first trial of fault injection. In this figure, the X axis shows the sequence of faults injected so far and each sequence of fault injections are labeled by the corresponding fault intensities. Y axis shows the number of possible correct key guesses corresponding to the each sequence of fault injections. The numerical data in Figure 4.2(bottom) is shown as a box-and-whisker diagram.

Figure 2(d) shows that strongly-biased faults (such as single-bit faults) leave less ambiguity about the correct key than weakly-biased faults (such as four-bit faults). The reason for this is simply a consequence of Hamming Distance computation over a byte: Under a single-bit error, a state byte can only map in 7 other byte values (of the possible 256). Under a two-bit error, however, a state byte can map in up to 55 other possible values.

In more elaborate fault injection campaigns, we will inject multiple faults, with varying intensity. Figure 4.2(b) shows the next level of the tree. In the second trial, the attacker can increase the fault intensity. Therefore, the second fault injection after single-bit could be two-bit, three bit or four-bit. Figure 4.2(e) shows the number of possible key guesses after post-processing the information of both trials. As an example, the number of possible key guesses after single-bit and two-bit fault injection is 7 in average. This number is smaller compared to injecting only single-bit (13 key guesses based on Figure 4.2(d)) or two-bit (33 key guesses based on Figure 4.2(d)) faults. The reason is that the attacker is now in possession of two sets of possible key guesses and can find the subscription of the two sets.

Finally, at the last level, the attacker injects the third fault (Figure 4.2(c)). Only one or two key guesses are left if the attacker has injected one, two and three faulty bits (Figure 4.2(f)). But there exists up to 11 key guesses if the attacker injects two, three and four bit faults. This demonstrates the fact that by injecting more accurate faults (single-bit), the key retrieval procedure takes less execution time and less trials. We also see that multiple fault injections quickly reduce the number of viable key candidates.

4.3.3 Experimental Setup

In order to evaluate our claims, we implemented an experimental setup as shown in Figure 4.3. The experimental setup consists of an FPGA (Altera Cyclone 4 E), a computer, and an external clock signal generator (Agilent 81110A Pulse/Pattern Generator). On the FPGA,

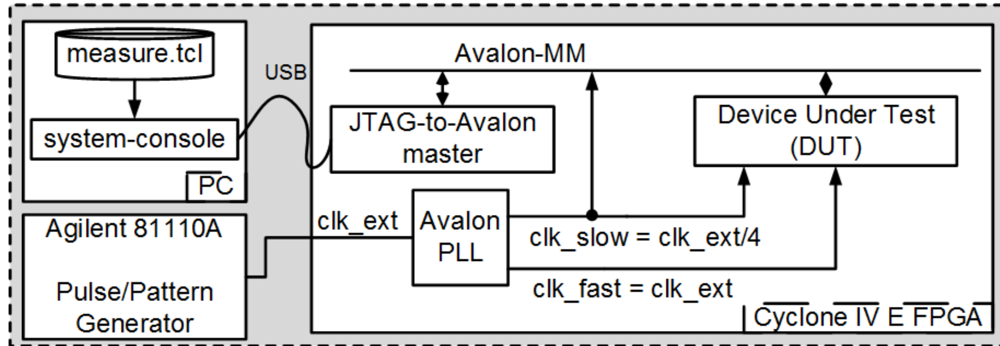


Figure 4.3: Block Diagram of the Experimental Setup

we have three blocks: Device under test (DUT), JTAG-to-Avalon master, and Avalon PLL. The DUT is connected to an Avalon-MM bus via Avalon memory-mapped slave interface. The DUT is controlled by a PC through a JTAG-to-Avalon master bridge. The PC uses an FPGA debug program (system-console) with a Tcl script (measure.tcl) in order to create read/write transactions on the Avalon bus. In our setup, we use clock glitches for fault injection and control the fault intensity by increasing/decreasing the frequency of the clock signal. Hence, the system has two clock signals *clk_fast* and *clk_slow*, which are derived from an external clock signal (*clk_ext*) using a phased-locked loop block (Avalon PLL). The external clock signal is generated by the external clock signal generator and it can take frequency values up to 330 MHz. The *clk_slow* signal drives the Avalon-MM bus and it is used for fault-free operation of the DUT. The frequency of the *clk_slow* signal is one fourth of the external clock signal frequency. The *clk_fast* signal is used to create clock glitches for fault injection and its frequency is equal to the external clock signal frequency.

Using the experimental setup with two different DUT blocks, we carried out two experiments:

1. In the first experiment, we investigate the behavior of four different S-box implementations by using them as DUTs in order to evaluate our biased fault model.

2. In the second experiment, we evaluate the DFIA attack on AES proposed in Section III by using an FPGA implementation of AES block cipher.

The details of the experimental setups will be explained in the following subsections.

4.3.4 Biased Fault Experiment for S-Box Architectures

In this experiment, we used four different FPGA implementations of AES S-box block as the DUT: PPRM1 S-box [33], Boyar-Peralta S-box [34], Canright S-box [35], and LUT-based S-box. In the DUT block of Figure 4.3, we place a combinational S-box implementation between a set of input/output registers as it is shown in Figure 4.4.

Our approach in this experiment is, for different external clock frequencies, applying a test input to the DUT, and then, computing the Hamming distance between the fault-free output and the test output corresponding to this test input. The fault-free output can be easily obtained by looking up the AES S-box table. The test output is the value obtained in the test output register of Figure 4.4 for the applied test input and clock frequency. Before applying the test input, we reset the test output register to the bitwise inverse of the fault-free output value via an initial input value. Hence, the Hamming distance between the fault-free and the test outputs gives the number of faulty bits obtained for the applied test input and clock frequency. Similar to the fault-free output value, the initial output value can be easily

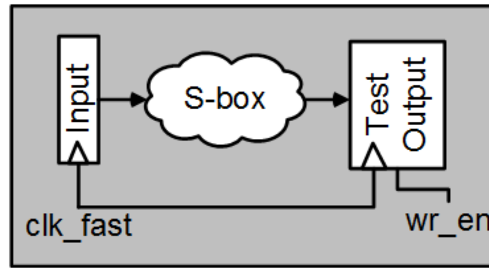


Figure 4.4: DUT Architecture for the Biased Fault Experiment

determined by looking up the inverse AES S-box table.

In our experiment, we arbitrarily selected 0x00 as the test input, and thus, the corresponding fault-free output value is 0x63. In order to reset the test output register to 0x9C (i.e. the inverse of 0x63) before applying the test input, our initial input value is 0x1C. A high-level timing diagram for the experiment is shown in Figure 4.5. The experiment starts with setting the input register to an initial input value to obtain the initial output value in the test output register. Then, the test input is applied to obtain the test output. These steps are controlled by a data write (*wr_data*) and an initialization (init) signal, which are generated by the Tcl script (measure.tcl) and are transferred to the DUT via JTAG-to-Avalon master block. Finally, the Hamming distance between the test and the fault-free outputs are computed.

In this experiment, we applied the above steps for four aforementioned S-box implementations and 33 different external clock frequency values. As a result, we obtained 33 experimental results for each S-box architecture. The experimental results are given in Figure 4.6. For each graph of Figure 4.6, the X axis represents the external clock frequency and the Y

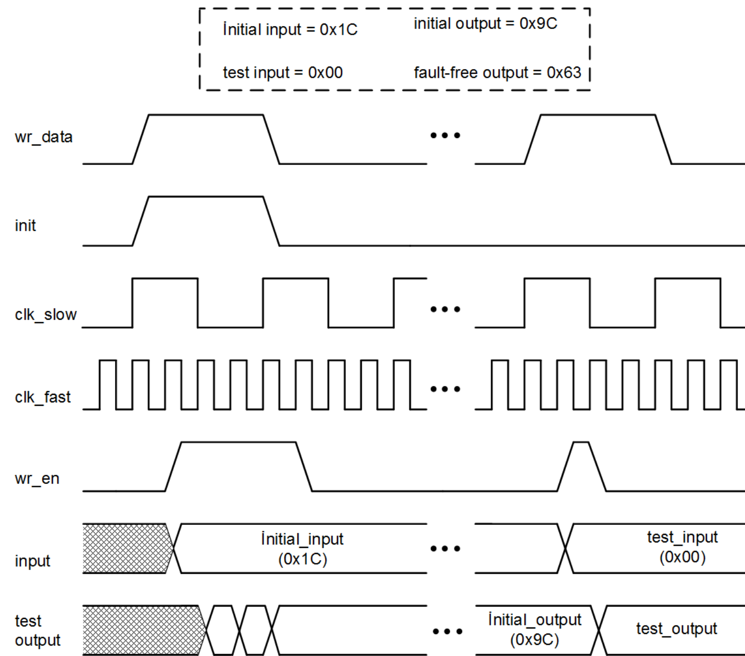


Figure 4.5: A High-level Timing Diagram for the Biased Fault Experiment

axis represents the Hamming distance between the fault-free and the test outputs. As it can be seen from Figure 4.6, the Hamming distance between the fault-free and the test outputs (i.e. the number of faulty output bits) increases with the clock frequency.

4.3.5 DFIA Experiment on AES

In this experiment, we evaluated the proposed attack on AES. We used the experimental setup given in Figure 4.3 with an AES block cipher implementation as the DUT. Our AES implementation is a register-transfer level Verilog definition of AES encryption and uses PPRM1 S-box architecture. In this experiment, our method is injecting biased faults into the AES state at 10-th round. In order to inject a fault into a position of the AES state, we

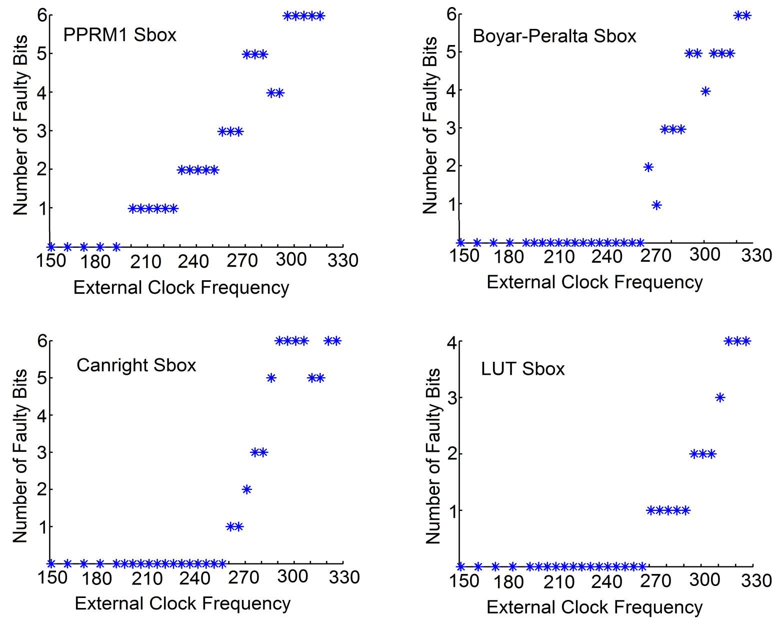


Figure 4.6: Biased Fault Behavior for Different S-box Implementations

use a fault-injection flip-flop (FI-FF) for this position rather than using a regular flip-flop (FF). As it is shown in Figure 4.7(a), the FI-FF includes a multiplexer and control logic in addition to a regular FF to select between two different clock signals. An FI-FF uses a high-frequency clock signal (*clk_fast*) at 9-th round of the AES, while using a low-frequency clock signal (*clk_slow*) at the other rounds. In Figure 4.7(b), we show a DUT architecture for fault injection into k-th byte of the AES state. In Figure 4.7, S[k] denotes the k-th byte of the AES state S, while S[k:n] denotes all of the AES state bytes from k-th byte to n-th byte. In this architecture, we replaced regular FFs used for k-th byte of the AES state with FI-FFs. Using this architecture, we can inject a clock glitch into the k-th byte of the AES state as it can be seen from the high-level timing diagram provided in Figure 4.8. In the DUT design, our assumption is that the fault injection affects only the state registers of

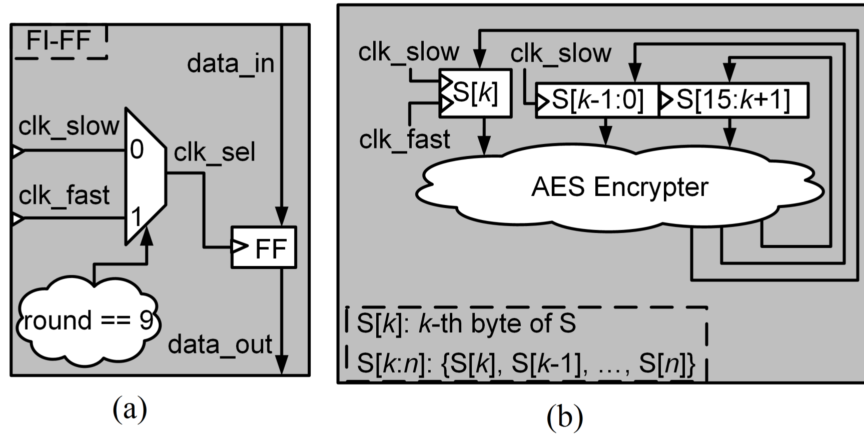


Figure 4.7: (a) Fault-injection Flip-flop (FI-FF) Architecture. (b) The DUT Architecture for Fault Injection into k-th Byte of the AES State.

an AES implementation. A reasonable extension to our method would be considering the effects of the fault injection on the other registers of AES as well. Hence, this is left as a future work.

In order to evaluate a key retrieval process by DFIA, we applied the attack on AES algorithm by different fault models. Hence, our experiment consists of two parts:

1. In the first part, we first injected faults with different intensities (i.e. for different external clock frequencies) into the most significant byte of the AES state ($S[15]$) in order to show how we retrieve a byte of the key using DFIA. Then, we injected faults into the separate bytes of the AES state ($S[14], S[13], ..., S[0]$) to compute how many single-byte fault injections are required, on the average, to retrieve the key via fault injections with single-byte granularity. We used the architecture given Figure 4.7(b) with different values of k ($k \in [15, 0]$) in this part of the experiment.

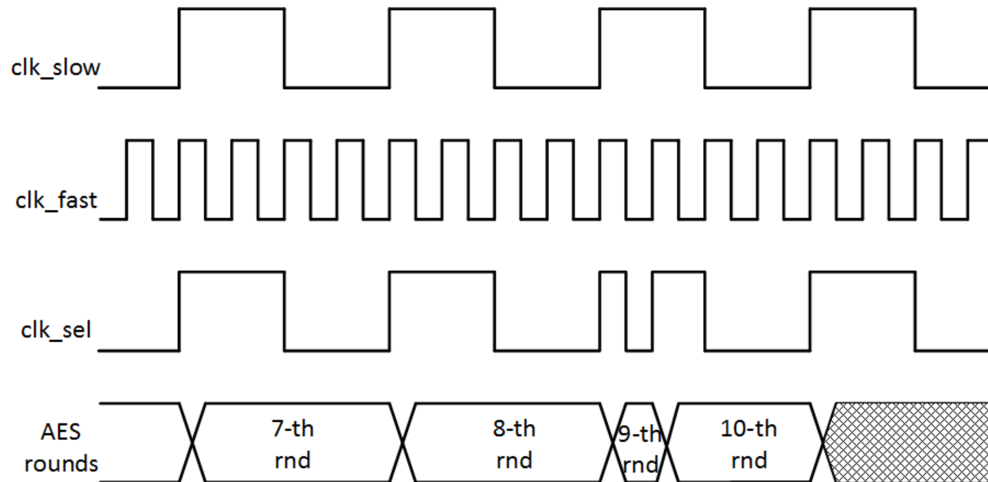


Figure 4.8: A High-level Timing Diagram for FI-FF

2. In the second part, we injected faults with different intensities into the whole AES state by changing all of the regular AES state FFs into FI-FFs. Our purpose in this experiment was to see how many fault injections with 16-byte granularity are needed to retrieve the key value.

In our experiment, we selected one arbitrary plaintext and an arbitrary key value. We obtained ciphertexts for fault injections with different intensities. Finally, we analyzed the results with a post-processing script executed in Matlab in order to find the total number of fault injections for a complete retrieval of the AES encryption key.

In this part, our assumption is that the attacker can affect only one byte of the AES state at 10-th round. In order to inject faults into separate bytes of the AES state, we used the architecture given Figure 4.7(b) with different values of k ($k \in [15, 0]$). The attacker can increase the external clock frequency and generate different faulty ciphertexts. Based

on results shown in Figure 4.6, the attacker is aware that the number of faulty bits is in proportion with the clock frequency. For each byte k of the AES state, we injected faults at four different intensities using different external clock frequencies: faultless, single-bit errors, two-bit errors, and three-bit errors.

Before proceeding to the results, we provide an example to show how we retrieve a single byte of the key using the experimental results: Figure 9 shows the results of the attack applied to the most significant byte of the AES state ($S[15]$). Based on the fault models, the Hamming Distance from the correct state to the faulty state should be one, two and three for the single-bit, two-bit and three-bit fault model respectively.

4.3.6 AES Key Retrieval Process with One Byte Fault Injection

Figure 4.9(a),(c),(e) plot the Hamming Distance between the fault-free state and the faulty state for one-bit, two-bit, and three-bit errors respectively. The X axis shows the key hypothesis, while the Y axis shows the Hamming Distance as computed with Table 4.2. Because of the varying fault intensity, we know that the correct key in the left graph can only have a Hamming distance of 1. The black dots show the possible key values. After post-processing the results, there are 14 possible keys (Figure 4.9(a), (b)). Hence, we conclude that a single-bit fault injection is insufficient to reveal the key. We therefore increase the fault intensity, and inject a two-bit error. The Hamming Distance graph now looks as in Figure 4.9(c). We expect the correct key is among those who show a Hamming Distance of 2. We can also see

that some key choices lead to a Hamming distance of only one. However, under two-bit fault injections, any wrong key choice will eventually end up at a Hamming Distances higher than 2, while the correct key will always remain at or lower than 2. We post-process the results of the one-bit fault injection and the two-bit fault injection by accumulating the Hamming Distance for each key guess. Again we find multiple candidates(Figure 4.9(d)). We increase the fault intensity once more, to three-bit errors. In this case, the correct key is among those with Hamming Distance at or lower than 3(Figure 4.10(a)).

We can now post-process the results of DFIA. The post-processing is simple: we accumulate the Hamming Distance for the respective keys for all graphs, and we look for the minimum. In this case, because we collected a graph at fault intensity 1, 2, and 3, we expect the minimum Hamming Distance in the overall graph to be at or lower than 6. Indeed, in the accumulated graph, we can find only a single key which shows a minimum Hamming Distance of 6, which allows us to conclude that this key is the correct one(Figure 4.10(b)).

It is important to note that the attacker might not be aware of the number of faulty bits in the state variable. Since the attacker increases the clock frequency for fault injection, he expects the number of faulty bits to increase. Therefore, for the key retrieval process, he still expects the correct key to be among the key guesses with the minimum summation of Hamming Distances.

Based on these results obtained for different bytes of the AES state, the average number of fault injections required for retrieving one byte of the key is 4.6. Furthermore, we need 68 fault injections to retrieve the whole AES key if the accuracy of the fault injection is one

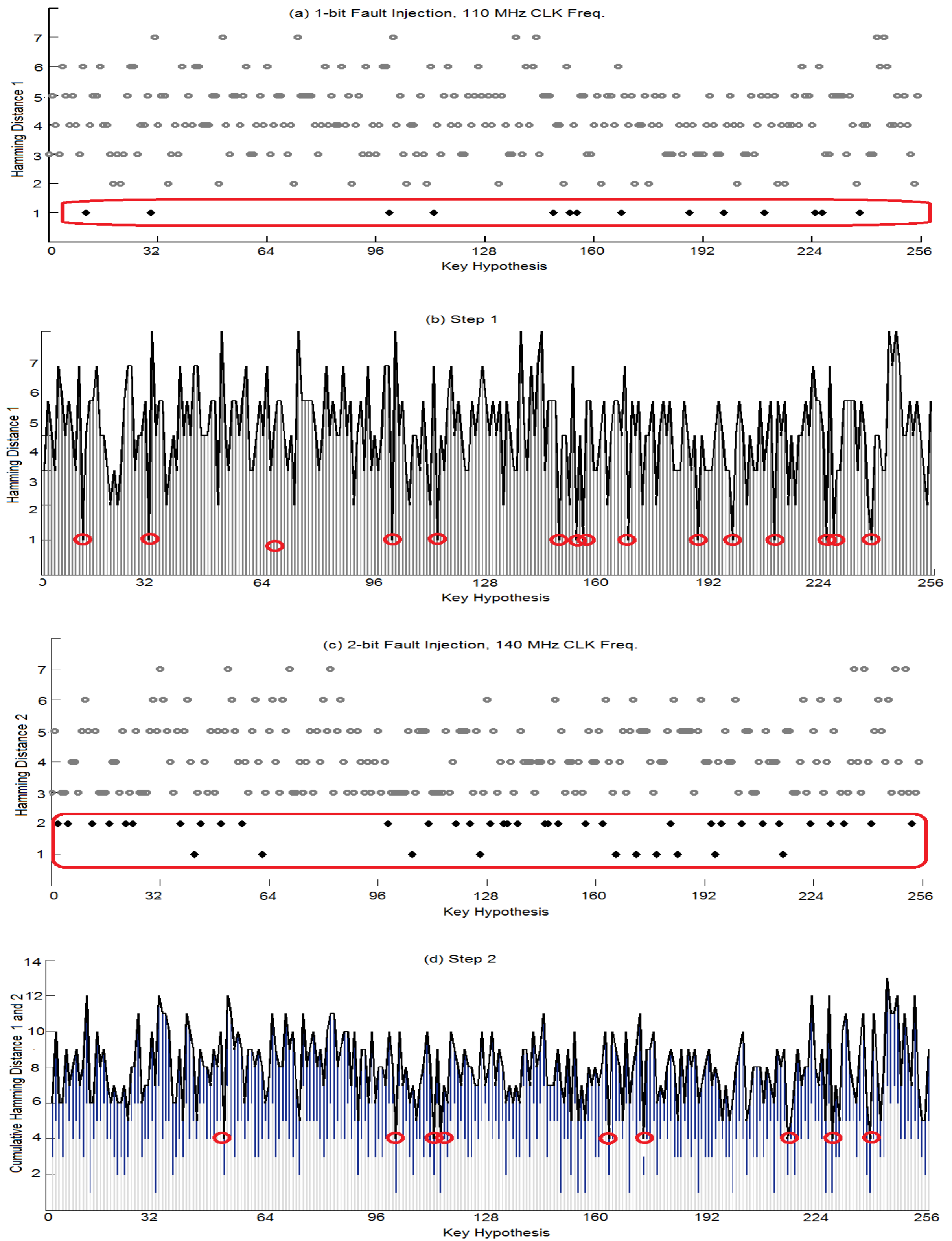


Figure 4.9: The Key Hypothesis Testing Algorithm Demonstration on a Measured Result, Step 1 and 2

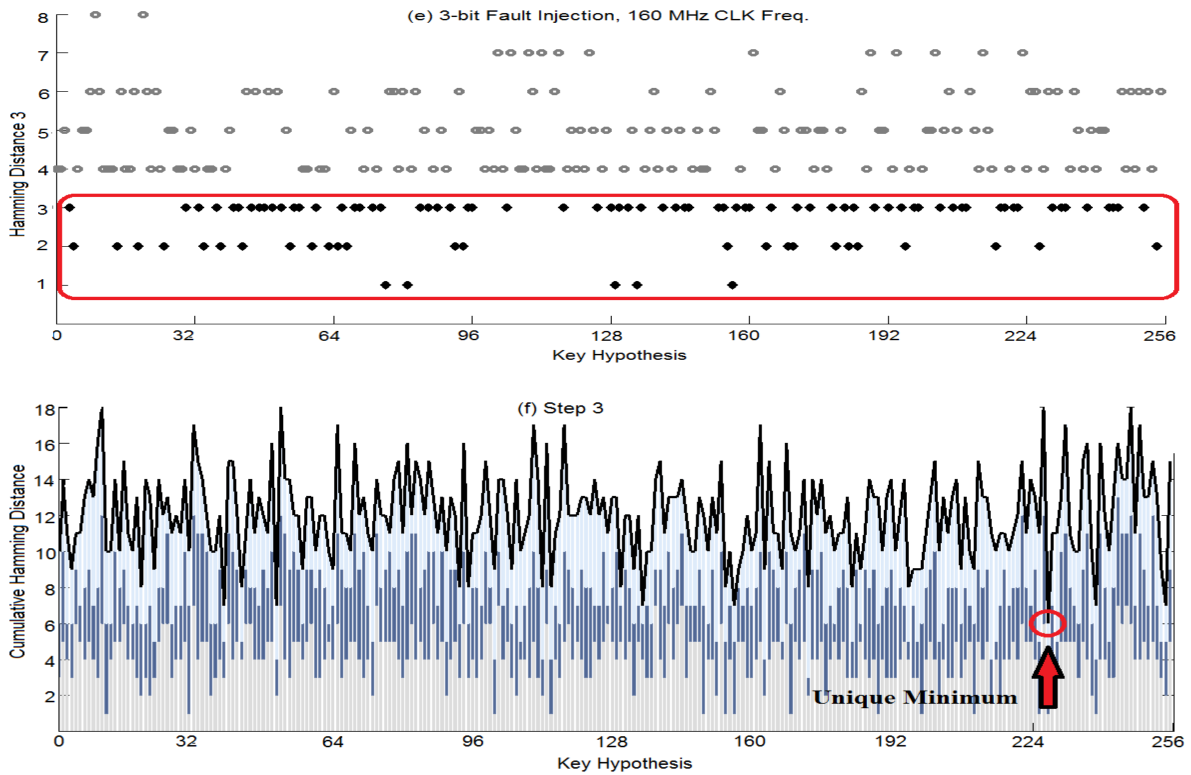


Figure 4.10: The Key Hypothesis Testing Algorithm Demonstration on a Measured Result, Step 3

state byte.

4.3.7 AES Key Retrieval Process with Multiple Byte Fault Injection

In this part, we injected faults into the whole AES state by changing all of the regular state FFs with FI-FFs in order to find the total number of fault injection with multiple byte fault injection for a complete key retrieval of the AES algorithm. We obtained ciphertexts for 24 different external clock frequencies, which are uniformly distributed from 100 MHz to 330

MHz. The results of this part shows that the whole AES key can be retrieved with 7 fault injections if the accuracy of the fault injection is whole state (i.e. 16 state bytes).

4.3.8 DFIA on PRESENT and LED

In this section, we explain the DFIA attack on nibble-serial and round-serial implementations of PRESENT and LED block ciphers. To get the full key, the attacker must perform DFIA for the last two rounds of PRESENT-80 (i.e. round 30 and 31) and the last three rounds of PRESENT-128 [36]. The LED cipher has a very simple key scheduling method, and thus, we can retrieve the key by attacking the last round of LED-64. For LED-128, we have to attack the last two rounds to retrieve the key.

DFIA has two phases: Injecting biased faults into the intermediate state of the block cipher and post-processing the faulty ciphertexts to retrieve the key. The biased fault injection is nibble-wise (i.e., 4-bit) for nibble-serial implementations, while it is state-wise (i.e., 64-bit) for round-serial implementations. Regardless of DFIA on round-serial or nibble-serial designs, the post-processing is always applied on a single key nibble at a time.

4.3.9 PRESENT Block Cipher

We make a brief overview of PRESENT and our implementations of it. PRESENT is a lightweight block cipher that was recently standardized by IEEE [37]. It uses an SP-network structure, and loosely follows the structure of AES, with the following important differences.

It has 31 rounds, and uses a block-length of 64 bits. It uses a selectable key size of 80 bit or 128 bit, and both versions are distinguished through their name (PRESENT-80 or PRESENT-128). Each round consists of three steps, including a roundkey addition layer, a nonlinear substitution layer with sixteen 4-bit Sbox, and a permutation layer. After the last round, an additional post-whitening step is included by adding a final roundkey.

The 64-bit roundkey is extracted from the upper part of the key register, and each round the key is updated with a key-size dependent key scheduling algorithm. The key schedule for PRESENT-80 is shown in Equations (1a) through (1c). The key schedule for PRESENT-128 is slightly more complex, and can be consulted in [37].

$$K_{79}K_{78}....K_0 = K_{18}K_{17}....K_{19} \tag{4.4a}$$

$$K_{79}K_{78}K_{77}K_{76} = Sbox[K_{79}K_{78}K_{77}K_{76}] \tag{4.4b}$$

$$K_{19}K_{18}K_{17}K_{16}K_{15} = K_{19}K_{18}K_{17}K_{16}K_{15} \oplus round_counter \tag{4.4c}$$

In this work, we studied both a round-serial and a nibble-serial implementation. The reason for this is to show the feasibility of DFIA on different implementations of the same cipher. The round-serial implementation computes an entire round of a complete block in a single clock cycle. This implementation is straightforward and follows the design of the original PRESENT paper [37]. We also developed a nibble-serial design, as shown in Fig. 4.11. In this case, one round for a single nibble (4 bits) from a block is computed in a single clock

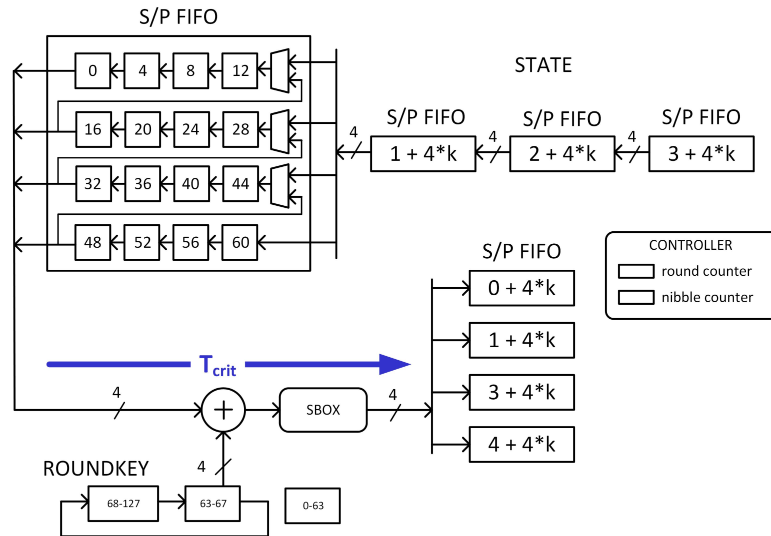


Figure 4.11: Nibble-serial Implementation of PRESENT

cycle, and this requires sequentialization of the round operations. This is easy to achieve for the roundkey addition and the Sbox substitution. For the permutation layer, we make use of the property that PRESENT's permutation is a 4-bit by 16-bit transpose operation: 4 bits of the permutation output are taken from a column of 4-bits of an input block, when the block is arranged as a 4-bit by 16-bit matrix. In Fig. 1, we implement the permutation using serial/parallel FIFO modules, which consist of four 4-bit FIFO's that either operate as a single 16-bit FIFO (serial mode) or else as four parallel 4-bit FIFO's (parallel mode). A complete block is stored in four serial/parallel FIFOs. Using two such structures, which store either the odd or even round states, a compact nibble-serial version of PRESENT is obtained.

Of particular note for our fault analysis is the critical path in these structures. The critical path runs through the Sbox and roundkey addition operations. For the round-serial design,

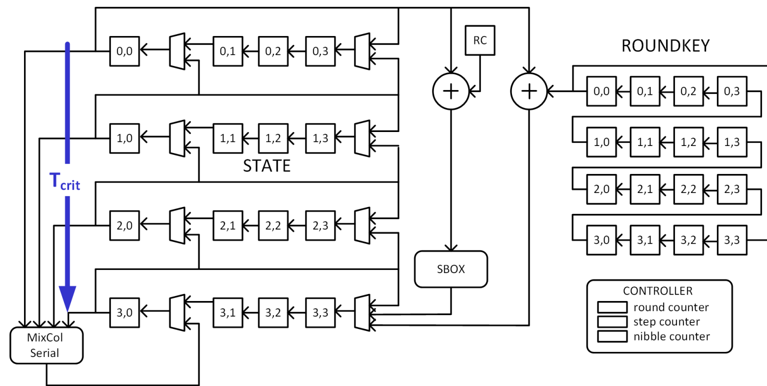


Figure 4.12: Nibble-serial Implementation of LED

all Sbox operations will be in the critical path in a given clock cycle. For the nibble-serial design, on the other hand, only a single Sbox operation will be in the critical path in a given clock cycle.

4.3.10 LED Block Cipher

The Light Encryption Device (LED) is a compact block cipher that was developed after PRESENT, and that integrates further insight into the lightweight cipher design process [38]. This block cipher, too, is an SPN structure, with a 64-bit block size. It supports two different key sizes, 64-bit or 128-bit, and the notation LED-64 and LED-128 is used to distinguish these cases. LED-64 has 8 steps of 4 rounds each, for a total of 32 rounds. In between steps, roundkeys are added. Each of the rounds includes operations similar to AES (AddConstants(AC), SubCells(Sbox), ShiftRows(SR), MixColumnSerial(MC)), but each of these steps is specifically optimized towards lightweight encryption. LED organizes the state as a four by four matrix of nibbles, and the round operations operate on these nibbles.

The LED cipher does not use a key scheduling algorithm. Rather, it reuses the same key for every step. In the case of 128-bit key, the key bits are divided into two groups and each round uses one of them alternatively. LED includes a post-whitening step with a final addroundkey.

As with PRESENT, we developed a round-serial and a nibble serial version of LED for DFIA analysis. Fig. 4.12 shows the architecture of the nibble-serial design. It follows the design guidelines of the original LED paper [38]. The State is organized in a FIFO-like structure of 16 nibbles. The structure can rotate the first column to compute MixColumnSerial, and it can rotate rows to compute ShiftRows. SubCells and AddRoundKey rotate the entire matrix through an Sbox and round-key addition respectively. The critical path runs through the MixColumnSerial. This is true for either the nibble-serial as well as the round-serial design. Fault injection using glitches will directly affect the variables computed in the critical path.

4.3.11 Implementations of the Block Ciphers

We wrote Verilog codes for our block cipher designs, namely, round-serial LED (LED-rs), nibble-serial LED (LED-ns), round-serial PRESENT (PRE-rs), and nibble-serial PRESENT (PRE-ns). We choose the key size as 128-bit in our implementations. We also generated gate-level netlist files for an Altera Cyclone IV FPGA (60nm Technology). We use these netlists for gate-level simulations, which are carried out using Modelsim-Altera 10.1d [39] software, to verify our claims throughout the chapter.

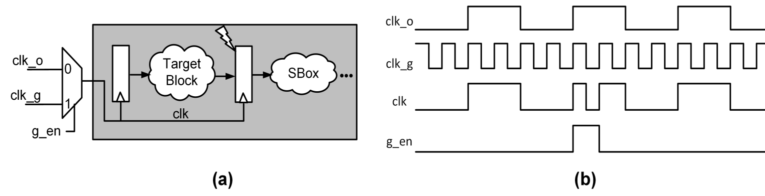
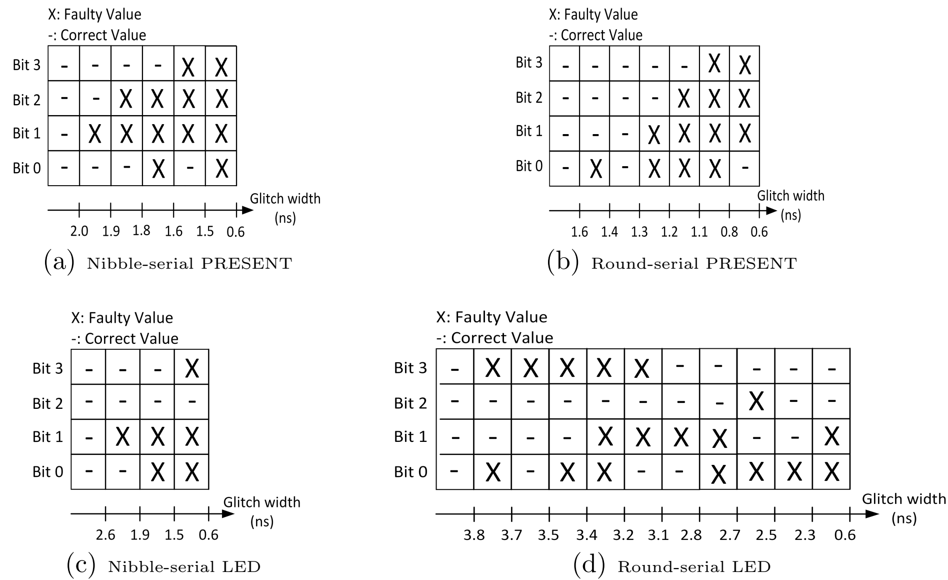


Figure 4.13: (a)Block Diagram of Experimental Setup (b)Timing Diagram of Experimental Setup

4.3.12 Biased Fault Injection in PRESENT and LED

The proposed DFIA attacks build upon injecting biased faults in the inputs of Sbox blocks. One can use a clock glitch injection method such as in Figure 3(a) for this purpose. This method generates an input clock signal for the circuit as a combination of two clock signals, namely, glitch clock (*clk_g*) and nominal clock (*clk_o*). As it is seen in Figure 3(b), we inject glitches in the *clk_o* via an enable signal (*g_en*). To inject a biased fault in the input of an Sbox, we set the *g_en* signal just before the clock cycle, in which the Sbox is employed. Such a glitch injection makes some timing paths fail and causes a biased fault in the input of the Sbox. We control the fault intensity by increasing/decreasing the frequency of the *clk_g* signal.

The target block of DFIA is different for each implementation. For LED-ns, the target block is MixColumnSerial (MC) logic. We create biased faults in the outputs of the MC logic by violating its timing paths. Then, the biased faults are transferred to the inputs of Sbox blocks via linear AddConstants (AC) layer. The target block of PRE-ns is the roundkey addition and substitution blocks. For LED-rs and PRE-rs the target blocks are the whole



(a) Nibble-serial PRESENT (b) Round-serial PRESENT (c) Nibble-serial LED (d) Round-serial LED

Figure 4.14: Biased Fault in the PRESENT and LED Implementations

round logic of the corresponding algorithms.

4.3.13 Biased Faults in PRESENT and LED Exist

In this section, we present a set of experimental results to verify that fault bias is a feasible fault source. We demonstrated biased faults through gate-level (post-place-and-route) simulation of the four block cipher implementations. In Fig. 4.14, we present the results for Sbox of the four implementations for a single plaintext.

Fig. 4.14 shows the relationship between the clock glitch width and obtained faults in the Sbox inputs at the last round of the corresponding implementation. For each subgraph of Fig. 4.14, the horizontal axis is the clock glitch width and the vertical axis is the bit position. We mark a faulty bit position with the symbol (X) and mark a fault-free position with

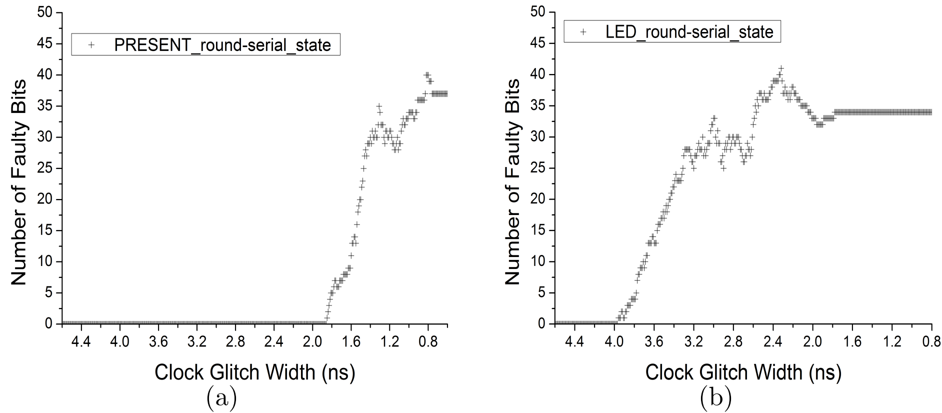


Figure 4.15: Biased Fault Injection on State of (a) PRESENT and (b) LED

the symbol (-). In each subgraph of Fig. 4.14, we observe a minimal Hamming Distance between two neighbor columns. This behavior verifies the existence of fault bias in our implementations.

In Fig. 4.15(a) and Fig. 4.15(b), we show the number of faulty bits that are induced in the 64-bit state with respect to the clock glitch width for PRE-rs and LED-rs, respectively. These two graphs show that the fault bias exists for the 64-bit state as well. The PRE-rs will fail at higher fault intensity (i.e. at a narrower glitch width) than LED-rs. The reason is that the critical path of PRE-rs is shorter compared to the LED-rs. Thus, the attacker needs higher-capability fault injection tool to inject fault into PRE-rs.

4.3.14 Post-Processing of DFIA on PRESENT

In this section, we describe the procedure to retrieve the key for PRE-ns and PRE-rs implementations. To obtain the 80-bit key of PRESENT-80, we first retrieve the round key of

round 31 to get the 64 most significant bits of the key. Then, to retrieve the remaining key bits, we retrieve the round key of round 30. Similarly, for PRESENT-128, the attacker must retrieve the round keys of rounds 31, 30, and 29.

We can retrieve each nibble of a round key separately. Therefore, the key retrieval procedure for nibble-serial and round-serial implementations is the same. Following is the procedure to retrieve the 80-bit key for PRESENT-80. We assume that the attacker has already collected the required amount of faulty ciphertexts to retrieve the key (using the method in Section 3.1).

The DFIA attack on PRESENT-80 follows Algorithm 2 as explained earlier. The faulty state variable is computed using Equation 2. By repeating this process for all nibbles of round 31, the attacker can retrieve the correct value for $K_{79}K_{78}....K_{16}$. In order to retrieve $K_{16}K_{15}....K_{0}$, the attacker has to process 4 least significant nibbles of round 30 as well.

$$S'_{k,C'} = PlayerInv(SboxInv(C' \oplus K)) \tag{4.5}$$

Fig. 4.16 shows an example DFIA attack to guess a nibble of a key. In this figure, the attacker injects four fault intensities: no injection, 1-bit fault injection, 2-bit fault injection and 3-bit fault injection. In this example, we retrieve one nibble of the round key with three fault injections. The bottom section of the bar chart shows the Hamming Distance between the first two intensities. The candidates for the correct key guess are the key guesses that show the minimum Hamming Distance, which is the set $G_1 = \{0, 2, 4, 7, 9, 10, 13, 15\}$ after the 1-bit

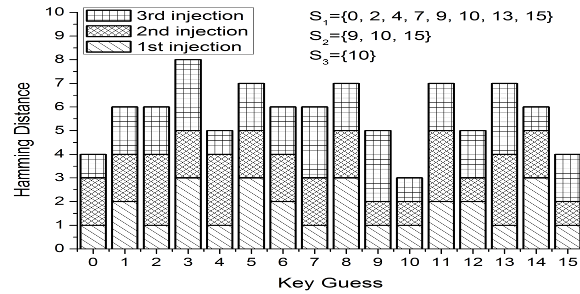


Figure 4.16: DFIA Steps to Retrieve 4-bit Key of PRESENT

fault injection. The middle section of the bar chart shows the Hamming Distance between 1-bit fault injection and 2-bit fault injection. As it is seen, the set of key candidates for correct key guess reduces to the set $G_2 = \{9, 10, 15\}$ after the 2-bit fault injection. The top section of the bar chart shows the Hamming Distance between 2-bit fault injection and 3-bit fault injection. The last fault injection gives us the unique key guess, which is $G_3 = \{10\}$.

4.3.15 Post-processing of DFIA on LED

In this section, we describe the procedure to retrieve the key for LED algorithm for nibble-serial and round-serial implementations. As the LED uses a very simple key scheduling method, the key can be retrieved by attacking the last round of LED-64. For LED-128, the attacker can retrieve the most significant 64-bit of the key by attacking the round 31. Then, he can retrieve the remaining bits by attacking the round 30.

The post-processing step of LED is different than the post-processing of PRESENT because LED includes a MixColumnSerial operation in its last round. The MixColumnSerial operation spreads the single faulty nibble in the intermediate state (S) to four nibbles of the faulty

ciphertext (C'). Therefore, the reconstruction of the hypothesized faulty intermediate state now requires a hypothesis on 16 key bits, which means that we have 2^{16} different hypotheses. We can solve this problem via a method proposed by Jeong et al. [40]. The solution relies on peeling off the MixColumnSerial operation of the last round by using an equivalent ciphertext ($C^{'*}$), and retrieving an equivalent key (K^*) instead of the actual key (K). The equivalent key and ciphertext satisfy the Equation 4.6a and 4.6b, respectively.

$$K^* = MCInv(K) \tag{4.6a}$$

$$C^{'*} = MCInv(C') \tag{4.6b}$$

As Equation 4.6b removes the effect of MixColumnSerial operation on C', one faulty nibble in S corresponds to one faulty nibble in $C^{'*}$. Therefore, we can use the $C^{'*}$ to retrieve each nibble of the K^* with 4-bit key hypotheses. Using S' and $C^{'*}$, we can perform DFIA (Algorithm 2) to retrieve four bits of the K^* using Equation 4.7. By repeating the procedure for 16 nibbles of the $C^{'*}$, we retrieve all 16 nibbles of the K^*. Then, we apply the MixColumnSerial operation on the K^* to retrieve the actual key K.

$$S^{'} = SboxInv(SRInv(C^{'*} \oplus K^{'*})) \tag{4.7}$$

The validity of the described solution can be seen from Equations 4.8a through 4.8d. The faulty ciphertext C' is computed by Equation 4.8a. Equation 4.8b is obtained by applying the MixColumnsInverse operation to both sides of Equation 4.8a. Using the distributive

property of the MixColumnsSerial over the XOR operation, we obtain Equation 4.8b. Using Equations 4.6b and 4.6a we obtain Equation 4.8d.

$$C' = MC(SR(Sbox(S'))) \oplus K \tag{4.8a}$$

$$MCInv(C') = MCInv(MC(SR(Sbox(S'))) \oplus K) \tag{4.8b}$$

$$MCInv(C') = MCInv(MC(SR(Sbox(S')))) \oplus MCInv(K) \tag{4.8c}$$

$$C'^{*} = SR(Sbox(S')) \oplus K^{*} \tag{4.8d}$$

4.3.16 Results of DFIA on PRESENT and LED

We evaluated the proposed DFIA attacks using gate-level simulation. In our gate-level simulations, we first generated 50 random plaintexts. Then, for each of the four implementations, we obtained the ciphertexts for different clock glitch widths. In this experiment, we gradually decreased the clock glitch width from $4.6ns$ to $0.6ns$ with $100ps$ step size. At the end, we obtained 40 ciphertexts for each plaintext and each implementation. As it can be seen from previous work, the selected step size is a reasonable value [41]. We present the analysis of our results in the following subsections. We also study the trade-off between glitch resolution and using multiple plaintexts in DFIA.

Table 4.3 shows the results of a DFIA attack on PRESENT and LED implementations. The first two columns in Table 4.3 show the maximum number of required fault intensity levels

Table 4.3: Required Number of Physical Fault Intensity Levels and Glitched Clock Cycles for DFIA Attack on PRESENT and LED with 100ps Fault Injection Resolution

	# of Fault Intensity Levels		# of Glitched Clock Cycles	
	Nibble-Serial	Round-Serial	Nibble-Serial	Round-Serial
PRESENT-80	10	12	160	12
PRESENT-128	16	18	256	18
LED-64	14	18	224	18
LED-128	28	36	448	36

for each implementation to get enough faulty ciphertexts in each nibble. For example, in nibble-serial PRESENT-80, the attacker is required to increase the fault intensity 10 times to get enough faulty ciphertexts in each nibble. The obtained numbers depend on the critical path of the target block for each implementation.

Column 3 and 4 in Table 4.3 show the maximum number of glitched clock cycles to obtain enough faulty ciphertexts for each implementation. As discussed in Section 2.3 and Section 2.4, in the nibble-serial implementation of the block ciphers, each nibble is processed in one clock cycle, while in the round-serial implementation of block ciphers, all 16 nibbles are processed at the same clock cycle. Thus, in the nibble-serial implementations, the attacker is required to inject clock glitch in 16 cycles to affect all nibbles, while in round-serial implementations, injecting the glitch in one cycle can affect all nibbles.

Compared to the previous fault attacks on PRESENT, we inject more faults. The attack

in [36] needs up to 150 faulty ciphertexts to retrieve the unique key. The attack proposed in [42] require 48 faulty ciphertext to retrieve the last round key of the algorithm. While the number of required fault injections in the DFIA attack is bigger compared to the mentioned previous works, we provide practical results with less restrictions of the fault model.

The previous DFA attacks on LED [40], requires a random faulty nibble to decrease the key search space to 2^8 candidates. Also, the methodology proposed in [43] is based on algebraic equations and injects a single fault to reduce the key search space to 26 $\sim$ 217 key guesses. The proposed DFIA attack on LED finds the unique correct key guess using additional fault injections. However, the biased fault model is practical and easy to achieve for the attacker.

4.4 DFIA with Multiple Plaintexts

This section summarizes Differential Fault Intensity Analysis. Algorithm 2 describes the attack procedure, and Table 4.1 lists the symbols used in this chapter. DFIA starts by applying a fault intensity q into an intermediate value S. The attacker next observes the faulty ciphertext C'_q, and derives the faulty intermediate value $S'_{k,q,P} = f(C'_q, k)$ under a key hypothesis k. The attacker repeats these two steps for Q different fault intensities by gradually increasing the fault intensity each time. In the post-processing step, for each key hypothesis, he computes the cumulative Hamming Distance among all faulty intermediate values. Finally, the attacker selects the key hypothesis that corresponds to the minimum cumulative Hamming Distance. The reason of looking for minimum is that for the correct

Algorithm 2 DFIA Attack Procedure using Multiple Plaintext

Assume *Cryptographic Algorithm, Fault Injection Tool*;

Result *Correct Key Guess* ;

foreach *Plaintext P* **do**

  foreach *Faultintensity q,* $1 \leq q \leq Q$ **do**

    Obtain faulty ciphertext C'_q;

    foreach *Key Hypothesis k* **do**

      Compute faulty state hypothesis $S'_{k,q,P} = f(C'_q, k)$;

    end

  end

end

//Post-processing phase ;

foreach *Key Hypothesis k* **do**

  Calculate $\rho_k = \sum_P \sum_{n=1}^{Q} \sum_{m=1}^{n-1} HD(S'_{k,n,P}, S'_{k,m,P})$;

end

$K = \min \rho_k$;

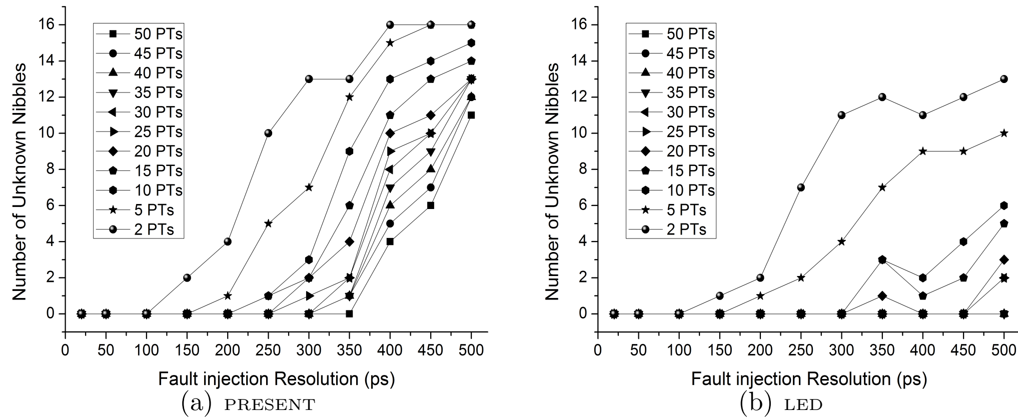
key hypothesis, the cumulative Hamming Distance is correlated with the fault intensity, and thus, it is minimal. A wrong key hypothesis infers a larger, random cumulative Hamming Distance due to the non-linear diffusion and confusion properties of the attacked cipher. Hence, the correct key results in the minimum cumulative Hamming Distance as long as

the applied fault intensities induce biased faults. Ghalaty et al. [44] show this behavior on AES for different biased fault injection scenarios. They draw two conclusions. First, DFIA converges for any given set of biased faults. Second, DFIA converges faster for strongly-biased faults (e.g. 1-bit faults) than it does for weakly-biased faults (e.g. 4-bit faults).

The original DFIA is applied using a single plaintext value [44]. However, DFIA can be easily extended to multiple plaintexts, by repeating the above steps for each plaintext, and by accumulating the resulting Hamming Distance values for each key hypothesis. Again, the global minimum will be obtained only under the correct key hypothesis. In this chapter, we make use of this feature, and we show that it can be used to improve the efficiency of DFIA when few biased faults are available, or when the fault injection equipment has limited resolution.

4.4.1 Trade-off between Fault Injection Resolution and Number of Plaintexts

In this section, we provide the experimental results to verify the efficiency of the extended version of DFIA. We investigate the relationship between fault injection resolution and the number of plaintexts that DFIA needs to retrieve the key. As our fault injection means is clock glitching, our fault injection resolution is the minimum increment or decrement in the clock glitch width that we can achieve. In this experiment, we apply DFIA attacks (Algorithm 1) on our PRE-rs and LED-ns implementations for different fault injection res-



(a) PRESENT (b) LED

Figure 4.17: Trade-off Between Fault Injection Resolution and Number of Plaintexts used for (a)PRESENT and (b)LED

olutions, from 20ps to 500ps, and for different number of plaintexts, which ranges from 2 to 50. Then, we count the number of the key nibbles that DFIA cannot retrieve under a given fault injection resolution and a given number of plaintexts. We call such nibbles as unknown nibbles throughout this section.

Fig. 4.17(a) and Fig. 4.17(b) present the results for PRE-rs and LED-rs implementations, respectively. In these figures, the Y axis shows the number of unknown nibbles out of 16 nibbles and the X axis shows clock glitch resolutions. Each data line in the graphs corresponds to a different number of applied plaintexts (PTs). Fig. 4.17(a) and Fig. 4.17(b) show two important behaviors for both LED-rs and PRE-rs. For a given fault injection resolution, using more plaintexts decreases the number of unknown nibbles. For a fixed number of plaintexts, the number of unknown nibbles decreases as the fault injection resolution increases (i.e., clock glitch step size decreases). An adversary can decrease the number of unknown nibbles either by increasing the fault injection resolution or by increasing the number of plaintexts

(i.e., encryptions). Therefore, we can conclude that there is a trade-off between the fault injection resolution and the number of required plaintexts. Due to this trade-off, DFIA can still efficiently retrieve the key when the fault injection equipment has a low resolution or when few biased faults are available.

Chapter 5

Analyzing the Efficiency of Biased-Fault Based Attacks

We will discuss and compare several recently proposed biased-fault attacks, including Fault Sensitivity Analysis (FSA, [3]), Non-Uniform Error Value Analysis (NEUVA, [32]), Non-Uniform Faulty Value Analysis (NUFVA, [45]) and Differential Fault Intensity Analsysis (DFIA, [2]). We will show that these four attacks share common ideas, and hence their performance can be compared. Performance, in the context of this chapter, is defined as the number of $(faults, plaintext)$ pairs needed to extract (fully or partially) the secret key of a cipher. For the experiments, we have selected *setup time violation* as the source of faults, and a hardware implementation of the Advanced Encryption Standard as the reference design. The faults are injected by controlled clock glitches. Our results are obtained from gate-level simulations, such that we are able to determine the exact cause of each fault. These are

pragmatic choices that make our results verifiable and comparable with other efforts.

5.1 Effects of Fault Bias on Circuit Behavior

In this section, we demonstrate the different effects of the fault bias on the circuit behavior, which are exploited by FSA, NUEVA, NUFVA, and DFIA. For this purpose, we provide some experimental results for two AES SBOX designs, namely, PPRM1-SBOX [33] and Comp-SBOX [46]. PPRM1-SBOX is a low-power SBOX design, which is based on a AND-XOR logic array. Comp-SBOX is a compact composite-field-based design, which decomposes the SBOX operation into smaller, lower-level finite-field operations.

Before proceeding further, we need the following definitions. We can model the effects of the fault injection on the targeted signal with an XOR operation:

$$s^* = s \oplus e \tag{5.1}$$

In Equation 5.1, the *correct value*, s, is the value of the targeted signal without fault injection. The *faulty value*, s^*, is the value of the targeted signal after fault injection. The *error value*, e, denotes the value of the injected fault itself. If *i-th* bit of the error value (e) is 1, the *i-th* bit of the correct value (s) is flipped and the faulty value (s^*) is obtained. Next, we will show different effects of fault bias on these values.

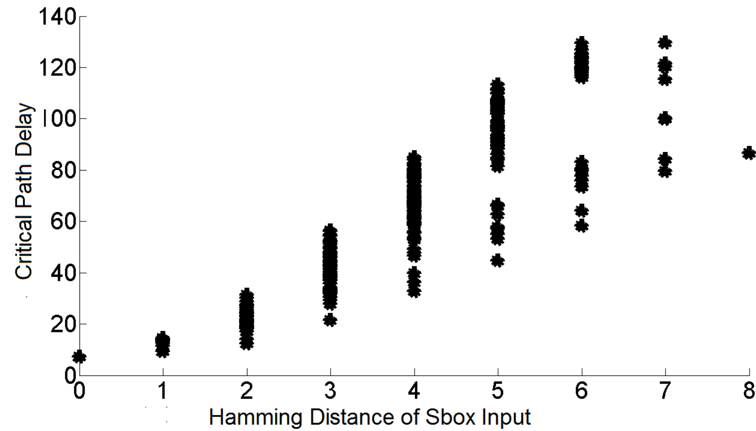


Figure 5.1: The relationship between the critical timing delay and the Hamming weight of the input of PPRM1-SBOX [1].

Data-Dependency of the Fault Sensitivity:

If an adversary gradually increases the fault intensity, a circuit can reach a point at which the output of the circuit becomes faulty. This threshold point is called *fault sensitivity*. For setup time violation, the critical path delay determines the fault sensitivity point. As the path delay distribution of a circuit is data-dependent, the fault sensitivity of the circuit is also data-dependent.

Ghalaty et al. experimentally demonstrated this effect on an FPGA implementation of PPRM1-SBOX architecture [1]. They obtained critical path delay value for each possible SBOX input in their experiment, where the initial values of SBOX outputs are *logic-0*. Figure 5.1 shows their critical path delay results with respect to the Hamming weight of the SBOX input values. As it is seen, the critical path delay of PPRM1-SBOX is proportional

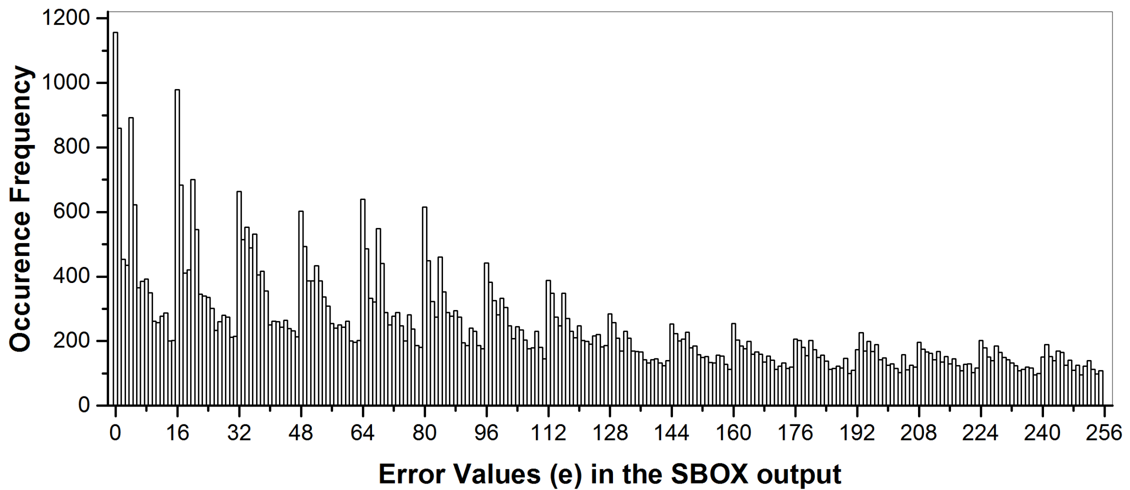


Figure 5.2: The distribution of injected error values in the output of PPRM1-SBOX ($Fault\ Intensity = 0.238GHz$).

to the Hamming weight of the SBOX inputs.

Non-uniform Error Value Distribution:

Fault bias can also cause a non-uniform distribution in the error value (i.e, fault pattern).

To illustrate this effect, we obtained error value distribution for a PPRM1-SBOX design. For this purpose, we applied a gate-level simulation for a post-place-and-route netlist, which is generated for a Xilinx Spartan6 FPGA. During the simulation, we applied all possible input transitions to the SBOX inputs and observed the error values injected into the SBOX outputs. We used a fault intensity of $0.238GHz$ in our experiment. Figure 5.2 shows the obtained error value distribution at the output of the SBOX. The horizontal axis shows the error values and the vertical axis shows the their frequency of occurrence. As it is seen, the

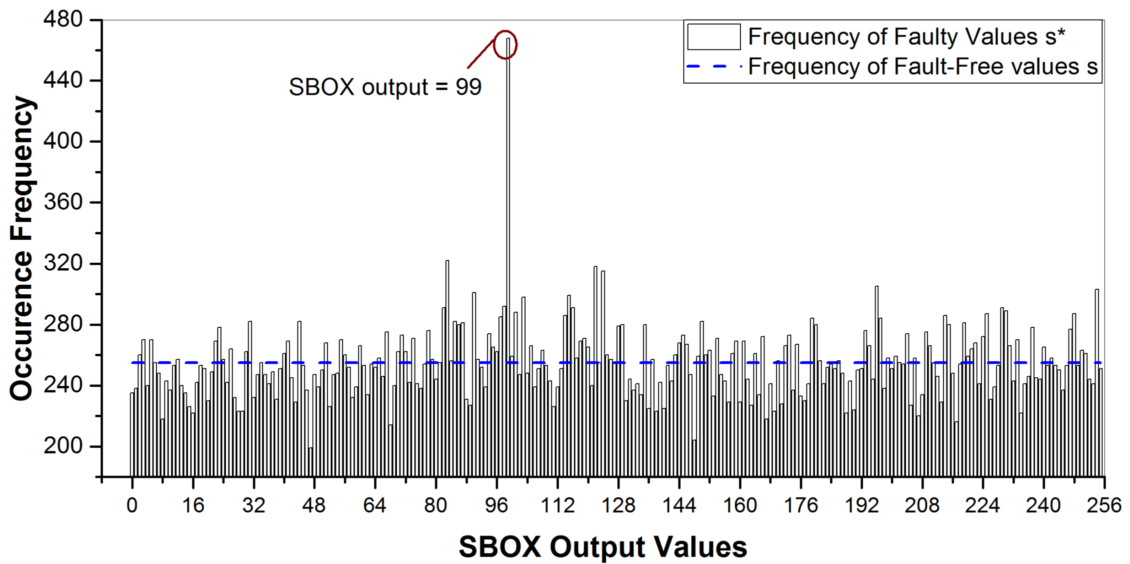


Figure 5.3: The distribution of faulty values in the output of Comp-SBOX ($FaultIntensity = 0.250GHz$).

fault bias causes a non-uniform error value distribution. Also, error values that have low Hamming-weight are more likely to occur for this fault intensity value.

Non-uniform Faulty Value Distribution:

Fault bias might also cause a non-uniform distribution in faulty values. By controlling the fault intensity, an adversary can make a circuit produce some faulty values more than the other faulty values.

Figure 5.3 demonstrates this effect for a gate-level netlist of Comp-SBOX, which is generated for a Xilinx Spartan6 FPGA. To obtain this figure, we applied a gate-level simulation with all possible SBOX input transitions and observed the faulty SBOX ouput values. Our fault

intensity was $0.250GHz$ in our simulation. The figure shows the distribution of faulty SBOX output values. The horizontal axis shows the faulty output values and the vertical axis shows the their frequency of occurrence. It also shows the distribution of fault-free SBOX output values. As it is shown (blue line of Fig. 5.3), the correct outputs of the SBOX have a uniform distribution. However, faulty outputs have a non-uniform distribution, and we see the output value 99 more than the other faulty values because of fault bias. This effect also experimentally demonstrated on an ASIC implementation of Comp-SBOX by Li et al. [47].

Small Changes in Fault Behavior

The effects of fault bias on a circuit's operation has also a differential character: A small change in the fault intensity will cause a small change in the fault bias. Therefore, an adversary can gradually increase the number of induced faults by gradually increasing the fault intensity. This enables the adversary to combine the information obtained at different fault intensities.

Ghalaty et al. experimentally demonstrated this effect on an FPGA implementation of PPRM1-SBOX architecture [2]. In their experiment, they collected the faulty SBOX output values, generated by a certain SBOX input value for 33 different clock frequencies. Figure 5.4 shows the number of faulty output bits, which they observed in the experiment, with respect to the applied clock frequency (i.e, fault intensity). As it is seen, the number of faulty output bits increases with the increasing fault intensity. As a consequence, this figure experimentally demonstrates the gradual fault behavior in proportion to the fault intensity.

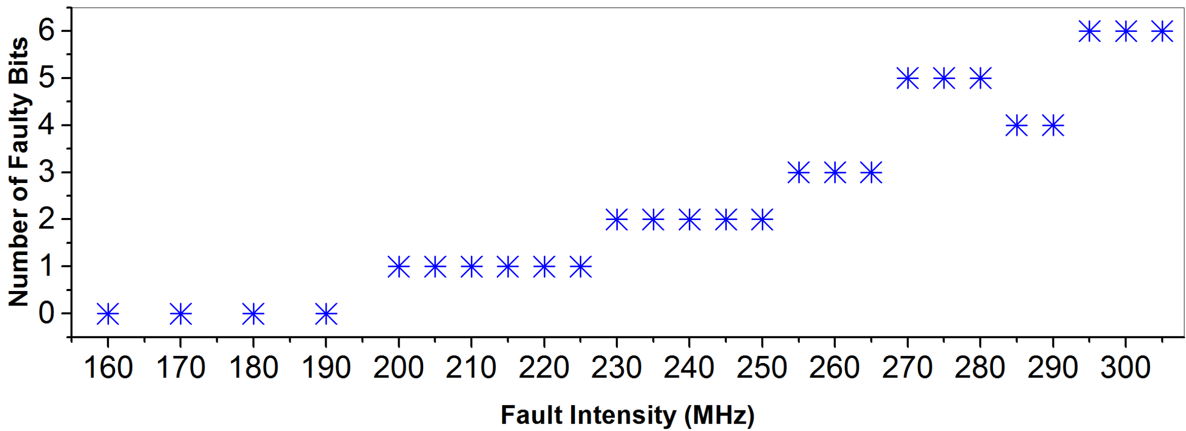


Figure 5.4: The relationship between the critical timing delay and the Hamming weight of the input of PPRM1-SBOX [2]

.

Next section provides a framework to build biased fault attacks, which rely on the effects of fault bias. Then, in Section 6, we will explain how FSA, NUEVA, NUFVA, and DFIA utilize the demonstrated fault bias effects.

5.2 Biased Based Fault Attacks

In this section, we explain FSA, NUEVA, NUFVA and DFIA on AES algorithm. Each attack is composed of 3 steps. These steps are explained in Chapter 4.1 (Figure 4.1).

5.2.1 Fault Sensitivity Analysis (FSA)

Fault Sensitivity Analysis (FSA) attack is proposed by Li et. al, in CHES 2010 [3]. Realizing that fault bias can have a data dependency on a secret value in a circuit, Li used it to demonstrate a fault attack with side-channel-like properties. The steps of FSA are as below.

- Step 1: In this attack, the target of the fault injection is round 10. The observables are plaintext, ciphertext and fault sensitivity points.
- Step 2: For FSA, the effect of fault bias is the data dependency of fault sensitivity. The adversary first inverts the ciphertext to round 10 input (S_{10}) using a key guess. Then, he estimates the effect of fault bias as the Hamming Weight of round 10 input $HW(S_{10})$.
- Step 3: In this step the attacker uses the Pearson Correlation Coefficient to find the key guess for which the fault sensitivity is strongly correlated to $HW(S_{10})$ for all inputs.

5.2.2 Non-Uniform Error Value Analysis (NUEVA)

Lashermes proposed a technique now abbreviated as Non Uniform Error Value Analysis, or NUEVA [32]. NUEVA relies on a biased distribution of error values. Lashermes used NUEVA in a differential evaluation technique. He evaluated the Shannon Entropy in the distribution of a secret error value under a given key hypothesis, and was thus able to distinguish a correct hypothesis from a wrong hypothesis. The steps of the attack are as below.

- Step 1: In this step, the target of fault injection is the output of round 9. The observables are correct and faulty ciphertexts.

- Step 2: Since, the effect of the fault bias in NUEVA is non-uniformity in error value, the adversary must estimate the fault bias on the error value. For this purpose, he inverts the faulty and correct ciphertexts for each key guess, and obtains the input of round 10 (S_{10}, S'_{10}). Then, the error value is estimated by XORing the S_{10} and S'_{10} values.

- Step 3: In this step, using the error values computed in Step 2, the adversary generates the error value distribution for each key guess. The adversary uses the Shannon Entropy to differentiate a key guess that has the strongest bias in the error value distribution.

5.2.3 Non-Uniform Faulty Value Attack (NUFVA)

Fuhr generalized the NUEVA technique, by directly considering the distribution of the faulty secret variable separately. His technique therefore is called Non Uniform Faulty Value Analysis (NUFVA), to indicate that the bias is present in the fault value itself, rather than in the error pattern [45]. He also proposed several distinguisher techniques. The NUFVA attack can be performed in different rounds of AES including round 7, 8 and 9.

- Step 1: We take the round 9 as the fault injection target. Since, this attack is a faulty ciphertext only attack, the observables include the fault intensity and the faulty

ciphertext.

- Step 2: The effect of fault bias for NUFVA is the non-uniform faulty value distribution. To extract the fault bias, the attacker must estimate the effect of fault bias on the faulty value. Therefore, he inverts the faulty ciphertext using a key hypothesis and obtain faulty inputs for round 10 (S'_{10}).

- Step 3: Using the values of faulty intermediate state, the attacker computes the distribution of faulty values for each key guess. Then, for each key guess, the attacker applies the Maximum likelihood function to distinguish the correct key guess from the wrong ones.

5.2.4 Differential Fault Intensity Analysis (DFIA)

The fourth technique that builds on fault analysis is Differential Fault Intensity Analysis (DFIA), proposed by Ghalaty [2]. The steps and properties of the attack are explained in the following steps.

- Step 1: The target of fault injection for DFIA is the output of round 9. The observables for this attack are the fault intensity and the faulty ciphertexts.

- Step 2: Unlike the previous techniques, DFIA does not assume that the fault distribution or the faulty value is biased. Rather, the fundamental difference with the previous techniques is that DFIA relies on small change in fault behavior as a result of small

change in fault intensity. To estimate the small change, the adversary computes the input of round 10 (S'_{10}, S''_{10},...), by inverting the faulty ciphertexts and key guess for several fault intensity levels. Then, he computes the distance between the hypothesized intermediate variables by using the Hamming Distance function.

- Step 3: The fault bias assumption for DFIA enables the use of a distinguisher that looks for the smallest change. Unlike the previous techniques, DFIA can combine fault behaviors collected at multiple fault intensities. Hence, the complete fault bias characteristic of a circuit can be exploited. Based on the assumption of fault attack, the error values are close to each other for the correct key guess. For wrong key guesses, the distance between injected error values will be random due to the non-uniform behavior of the Sbox module. Therefore, the distinguisher function simply chooses the key that shows the minimal distance between intermediate variables.

5.3 Experimental Setup

In this work, we injected biased faults into a device under test (DUT) through gate-level simulation. As the DUT, we use two AES-128 designs: PPRM1-SBOX-based AES-128 (PPRM1-AES) and Comp-SBOX-based AES-128 (Comp-AES). We generated the gate-level netlists of these designs for a Xilinx Spartan6 FPGA (45nm technology). Both DUTs compute each round of AES in a separate clock cycle. We use clock glitches as the fault injection means (Fig. 5.5(a)). This method generates a clock signal for the circuit as a combination

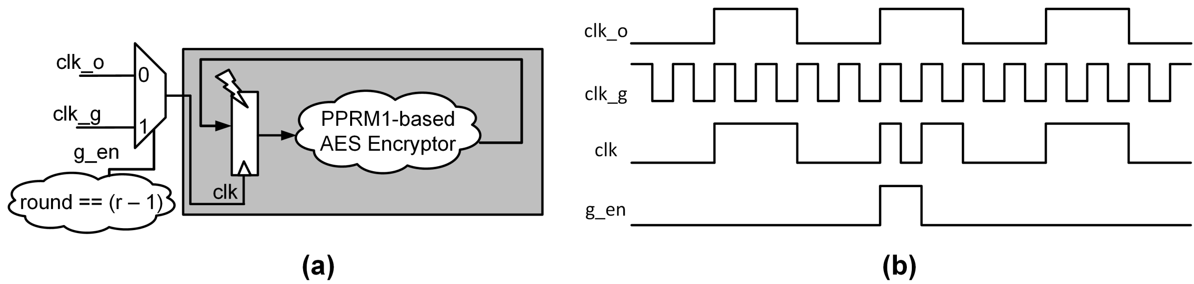


Figure 5.5: **(a)** Block diagram for the experimental setup. **(b)** Timing diagram for the experimental setup.

of two clock signals, namely, glitch clock (*clk_g*) and nominal clock (*clk_o*). As it is seen in Figure 5.5(b), we inject glitches in the clk_o via an *enable signal* (*g_en*). To inject a biased fault in the input of the *r-th round* of AES, we set the *g_en* signal just before the clock cycle, in which the *r-th round* is computed. Such a glitch injection makes some timing paths fail during $(r-1)$-*th round* and causes a biased fault in the input of *r-th round*. We control the fault intensity by increasing/decreasing the period of the *clk_g* signal.

As each considered attack has different requirements for the fault injection, we collected a large set of fault injections results to compare their performances. We repeated the following steps in our gate-level simulations for each DUT. We first generated 1000 random plaintexts. Then, for the *rounds 6-10* of AES, we obtained the ciphertexts for different clock glitch periods. In this experiment, we gradually decreased the clock glitch period from 16ns to 0.6ns with 100ps step size. At the end, we obtained 154 ciphertexts for each plaintext and 154000 ciphertexts for each round. In Section 8, we will use these ciphertexts to evaluate the performances of the considered biased fault attacks.

5.4 Efficiency Analysis of Biased Based Fault Attacks

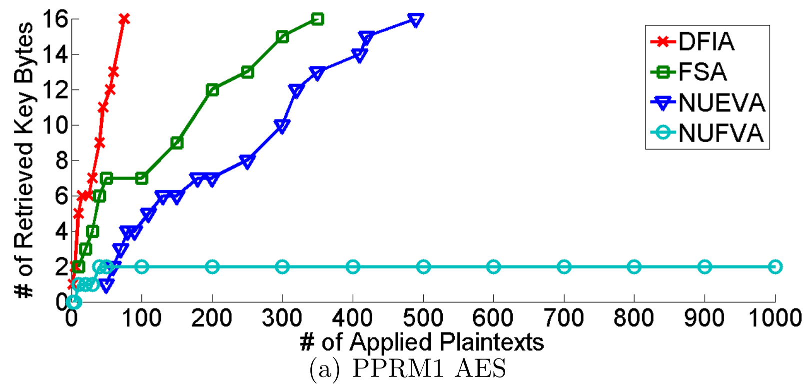
In this section, we show our results for two fault injection conditions. The first case is the ideal condition, in which the target of the fault injection is a specific round of the AES algorithm.

In the second condition, we assume that the fault injection is in a noisy environment or the adversary is not able to control the timing of the glitch injection precisely [48]. Some previous works show that in case of using other injection tools such as Electromagnetic pulse injection, the attacker might not be able to specifically inject fault into one round [48]. In this case, we assume that the faults might occur in other rounds of the AES algorithm. In this case, we randomly choose the faulty results from several rounds of AES and study the effect of noise in the performance of the attacks.

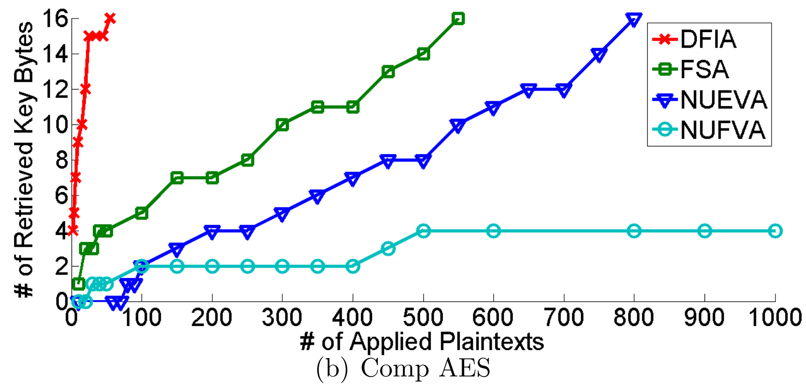
5.4.1 Results for Ideal Fault Injection

In the ideal condition, we assume that the fault injection tool is based on the clock glitching and the attacker is able to identify the location of the fault in the AES algorithm. Based on the requirements of the discussed attacks, we inject faults in Round 9 for DFIA, NUEVA and NUFVA, and in round 10 for the FSA algorithm, in order to retrieve the key of the last round. Figure 5.6 shows the results of applying different attack strategies on two implementations of AES algorithm (PPRM1-AES and Comp-AES).

The first fault injection strategy is by starting from the correct ciphertext. Then, we grad-



(a) PPRM1 AES



(b) Comp AES

Figure 5.6: Number of Required Fault Injection Attempts to Retrieve the Key with Different Attack Strategies in Ideal Condition (a)(b)with Fault Sensitivity Information

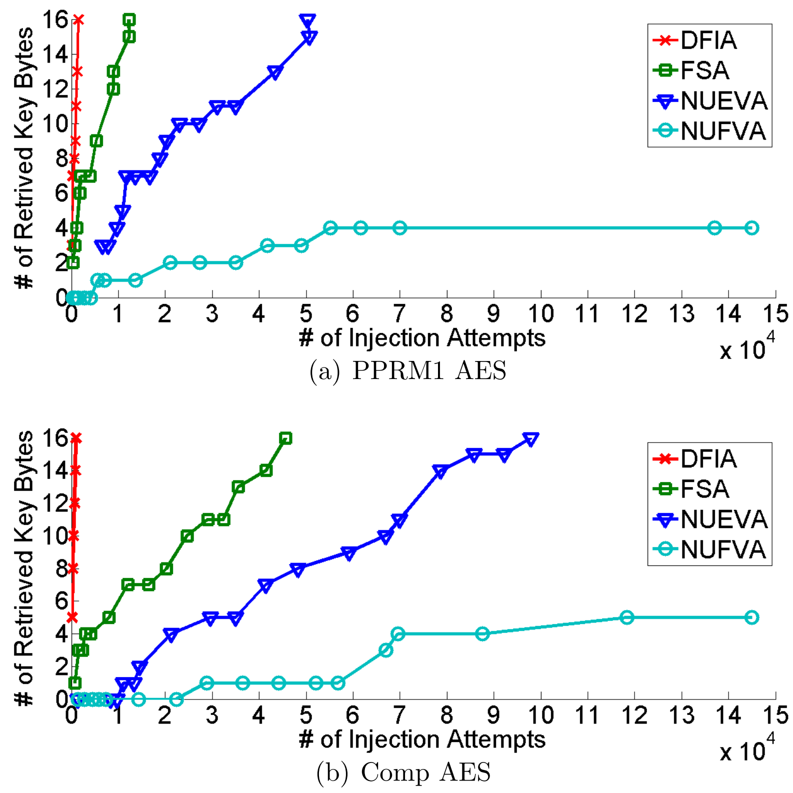
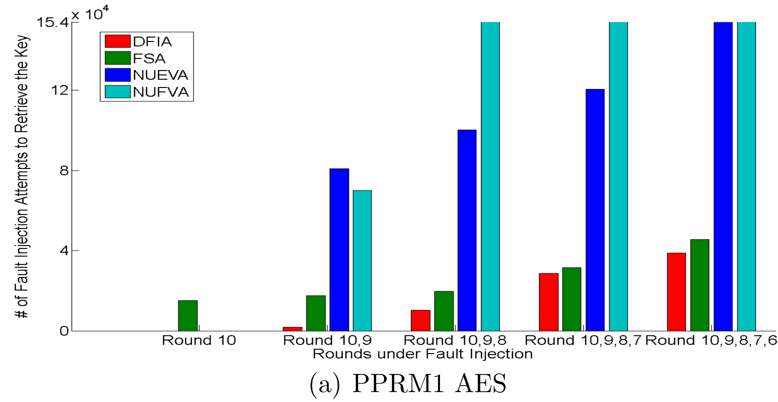


Figure 5.7: Number of Required Fault Injection Attempts to Retrieve the Key with Different Attack Strategies in Ideal Condition (a)(b)with All Possible Fault Intensities

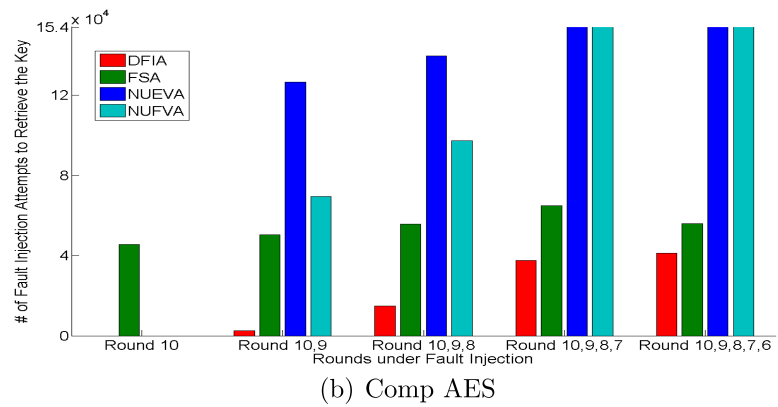
ually increase the fault intensity until we observe the first faulty ciphertext at the fault sensitivity point. We captured the faulty ciphertext with this method for 1000 plaintext. Then, we applied four attack strategies to the set of faulty ciphertexts. The results in Figure 5.6(a) and 5.6(b) show the required number of plaintext for retrieving the key. In this case, the required number of plaintexts simply shows the number of fault injection attempts, since there s one fault sensitivity point for each plaintext. As shown, in this case, even if we do not have multiple fault intensities, the DFIA attack works with less number of plaintexts compared to FSA and NUEVA. The NUFVA attack is not able to retrieve all bytes of the key as observing stuck-at or biased faulty value with this method of fault injection is very difficult.

The second fault injection methodology is the extension of the first one. Starting from the correct ciphertext, we gradually increase the fault intensity for each plaintext and keep different faulty ciphertexts for each plaintext. We injected 154 levels of fault intensity for 1000 plaintext. Then, we applied four different attacks on these faulty ciphertexts. Each attack uses a certain amount of the injected faults based on its requirements. For example, in FSA attack, the attacker only requires the fault injections up to the fault sensitivity point. Or for DFIA, the adversary requires increasing the fault intensity up to the point of generating the last faulty byte. For each attack, we count the number of useful fault intensity levels associated with each plaintext to find the total number of fault injection attempts.

Figure 5.7(a) and 5.7(b) show the results for the second fault injection methodology. The results show that the DFIA attack can retrieve the key efficiently with using less than 2000



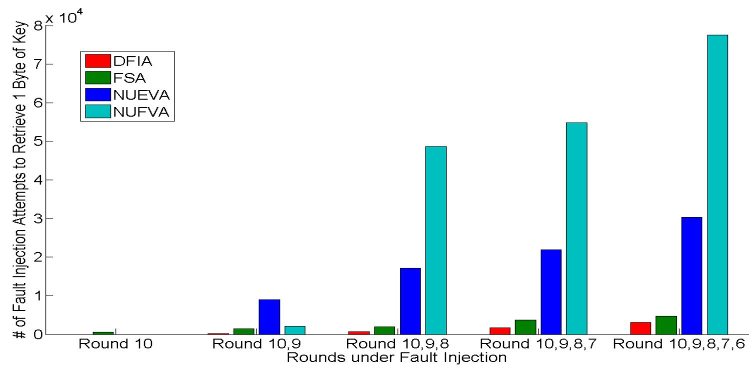
(a) PPRM1 AES



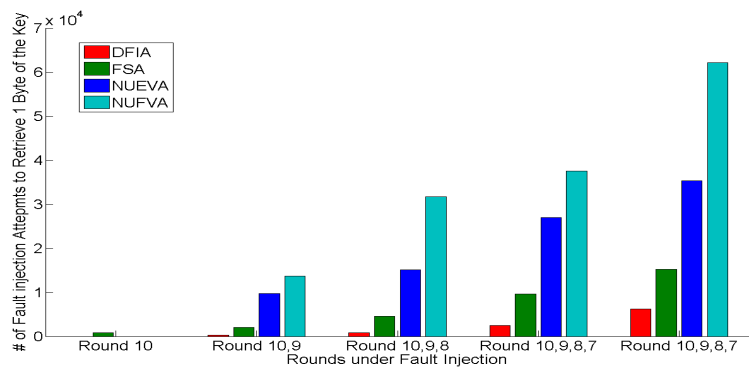
(b) Comp AES

Figure 5.8: Number of Required Fault Injection Attempts to Retrieve the Key with Different Attack Strategies in Noisy Environment. Each group shows the rounds that are affected by the fault injection.

fault injection attempts. The number of fault injection attempts used by the FSA in Comp-AES increases because observing the effect of fault bias is more difficult in Comp-SBOX implementation. NUFVA attack can only retrieve less than 5 bytes of the key. The reason is that we cannot observe biased faulty output or stuck-at faults in the intermediate state.



(a) PPRM1 AES



(b) Comp AES

Figure 5.9: Number of Required Fault Injection Attempts to Retrieve one Byte of the Key with Different Attack Strategies in Noisy Environment. Each group shows the rounds that are affected by the fault injection.

5.4.2 Results for Noisy Fault Injection

Assuming a noisy fault injection environment, we injected faults into round 6, 7, 8, 9 and 10. Then, as the set of faulty values, we choose faults from each round randomly. Then, we apply the four fault attacks, and count the number of fault injection attempts it requires to find the key. Figure 5.8(a) and 5.8(b) shows the number of required fault injections for each case. Since NUFVA attack cannot fully retrieve the key, the number of fault injection attempts shown is for the maximum number of key bytes that it can retrieve. As shown, DFIA is able to retrieve the key by less number of attempts compared to other attacks. The number of fault injection attempts increases exponentially for FSA attack. The reason is that due to the noise in the induced fault, the fault sensitivity point associated with each plaintext is for the data of the noisy rounds, rather than round 10. The NUEVA attack, is still successful when the noise is in round 8, however, in the last case, NUEVA can only retrieve 9 bytes of the key using all available plaintexts.

Since, NUFVA is not able to fully retrieve the key, to provide a fair comparison of the attacks, Figure 5.9(b) and 5.9(b) show the number of required fault injection attempts to retrieve only one byte of the key. As shown in these figures, the required number of attempts for DFIA is much less compared to other attacks.

Table 5.1: Cost of Fault Attacks for Noisy and Ideal Fault Injection Conditions

	DFIA	FSA	NUEVA	NUFVA
Ideal-PPRM1	1518^{-1}	12250^{-1}	50320^{-1}	69000^{-1}
Noisy-PPRM1	10346^{-1}	19800^{-1}	100100^{-1}	154000^{-1}
Ideal-Comp	980^{-1}	45650^{-1}	91650^{-1}	112800^{-1}
Noisy-Comp	14800^{-1}	55610^{-1}	139650^{-1}	154000^{-1}

5.4.3 Attack Efficiency

In this section, we intend to compare the efficiency of the biased fault based attacks. The cost of an attack can be defined by the number of fault injections and the number of applied plaintexts used for the attack. As mentioned in previous sections, for each applied plaintext, we count the number of useful fault injection attempts. The attack efficiency in this chapter is defined with the Equation 5.2.

$$Attack\ Efficiency = (\#Plaintexts \times \#Fault\ Injection\ Attempts\ per\ Plaintext)^{-1} \tag{5.2}$$

To compare the attacks, we counted the number of fault injections and number of applied plaintext for each attack in two conditions. First is the ideal condition that we injected fault only in the target location and second is the noisy condition in which we injected the fault in rounds 8, 9 and 10. Table 5.1 shows the efficiency of different fault attacks, in these two conditions for two implementations of AES. Since, NUFVA cannot completely retrieve the

key, we provided the minimum number of fault injection attempts that it uses for partial key retrieval. Based on the results, DFIA needs less number of fault injection attempts compared to other attacks. The reason is that, the assumption on the effects of fault bias for the DFIA attack benefits from combining multiple fault intensity levels per plaintext. This property maximizes the information we can obtain from each biased fault injection and hence, helps to improve the efficiency of the DFIA attack.

5.5 Conclusion

In this chapter, we presented a comparison of four recently published attacks that use fault bias, the non-uniform response of digital systems towards fault injection. By investigating the common elements of these attacks,we were able to build a single framework that supports a systematic comparison. Our main conclusion is that, even though all of the investigated attacks use the same test case of glitch injection on an AES design, their performance differs greatly. Using the number of fault injections as the cost metric, we found that DFIA performs best, followed by FSA, NUEVA and NUFVA. The main reason for the better performance of DFIA is the differential nature of the analysis mechanism. This makes DFIA more tolerant against estimation mistakes and noise effects. Also, DFIA can use the entire fault characteristic for a given input stimulus, in contrast to other fault injection techniques, which uses only a single fault per input stimulus. Overall, fault-bias based techniques are effective as an implementation attack, and they show that fault-injection based attacks can

be applied in a generic setting with minimal assumptions on the underlying cryptographic implementation. It seems reasonable to conclude that there is a rapidly increasing need to develop countermeasures against fault bias in digital hardware, including firmware-driven embedded systems.

Chapter 6

A Design-Time Countermeasure against Fault-Sensitivity Analysis

In this chapter, we analyze the causes of FSA attack by analyzing different S-box architectures [3]. We demonstrate the existence of two factors, the depth of the AND/OR network in a design, as well as the arrival time of input signals to the AND/OR network. Both of these factors influence the fault sensitivity. Based on these two factors, a countermeasure has been suggested to eliminate fault sensitivity in a design based on a delay insertion algorithm which could be optimized in future works for area minimization. In this chapter, we first review the related works on recently proposed countermeasures against fault attacks, and then analyze the causes of FSA to propose a delay based countermeasure.

6.1 Related Work

This section reviews previous and related work in FSA. It has been shown that most of the countermeasures against the Side Channel Attacks and fault attacks are not efficient against FSA. The authors of [49] have shown that the WDDL design is DFA resistant. But, it has been shown that this design is vulnerable against FSA [50]. In FSA, since no faulty output value is required, the attack is also resistant against the fault prevention methods employed against the DFA attack. An example of this is the AES module equipped with the concurrent error detection schemes [51] that has been broken in [52].

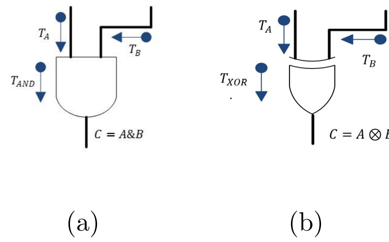
Certain masking techniques have been also broken by FSA. Based on [53], when the masked values are based on random numbers, the faulty ciphertext would have a non-uniform distribution and can be correlated with the input values. The AES module with the Masked AND Operation (MAO) has been broken by [53].

The first proposed countermeasure against the FSA is [54]. Earlier work, namely [55], proposed the use of an enable signal to eliminate the data dependency of fault sensitivity. The results of the combinational logic are stored until the timing of the enable signal arrives. However, Li did not clarify how to generate this enable signal [55]. The authors of [54] suggested a solution for this. A one-time memory stores the timing of the combinational logic. The Delay Blocks are reconfigured based on the values of the one-time memory and set the enable signal. While the area cost of their method is 10%, their method is a post-manufacturing reconfiguration which is technology dependent. Moreover, they use an external module for

generating the delay blocks based on the circuit delay. This external module is a multiplexer that decides on the number of inserted delay elements. An attacker can tap on the external module or specifically the multiplexer select signal to get the fault sensitivity information. In contrast, our proposed method is based on changing the circuit internally.

6.2 Fault Sensitivity Analysis

The attack procedure in FSA exploits the data dependency of fault sensitivity. In this chapter, we are specifically interested in analyzing the cause of fault sensitivity. Li observed that gates become fault sensitive when their inputs have a different arrival time [3]. Figure 6.1(a) illustrates the case of an AND gate. If we assume that $T_A < T_B$ (which means that signal B has gone through a longer path than signal A), then T_C depends on the value of A. If $A = 0$, then $T_C = T_A + T_{AND}$. In other words, if A has the known value zero, then the arrival time for the AND gate output is defined by the arrival time of A plus a small constant delay determined by the AND gate. In other other case, when $A = 1$, the A input does not affect the eventual value of the output, and any transition on C is defined by transitions on the input B. Therefore, $T_C = T_B + T_{AND}$. Hence, we conclude that an AND-gate is fault-sensitive: its switching time depends on the value of an input bit A. The same happens for the OR gate, and it is illustrated in Table 6.1. If $A = 1$, then $T_C = T_A + T_{OR}$. Otherwise $T_C = T_B + T_{OR}$. In contrast to the AND and OR gate, the XOR gate is not fault sensitive. For XOR gates (Figure 6.1(b)), the output C will propagate changes to either A or B with



(a) (b)

Figure 6.1: Data Dependency of Fault Sensitivity for Different Gates [3]

Table 6.1: Data Dependency of Fault Sensitivity in AND, OR and XOR Gates

	A	B	T_C
AND	0	x	$T_C = T_A + T_{AND}$
AND	1	x	$T_C = max(T_A, T_B) + T_{AND}$
OR	1	x	$T_C = T_A + T_{OR}$
OR	0	x	$T_C = max(T_A, T_B) + T_{OR}$
XOR	x	x	$T_C = max(T_A, T_B) + T_{XOR}$

the same preference. Therefore, $T_C = max(T_A, T_B) + T_{XOR}$. Table 6.1 summarizes the above arguments. In the remainder of the chapter, we use the term **Effective Gates** to indicate the gates that are fault sensitive such as AND and OR gates.

The FSA attack has two phases, the fault sensitivity information collection, and the key retrieval procedure. In the first phase, the attacker applies a plaintext as an input to the cryptography device, then gradually increases the fault intensity until he sees some abnormality in the output of the device. This point is recorded for the applied plaintext as the

fault sensitivity. The profiling phase is performed for N different inputs. In the key retrieval phase, the attacker has a key guess, the ciphertexts and the fault sensitivity for the profiling input. The attacker finds the fault sensitivity for the potential key guess. Then, he draws the correlation graph between the actual fault sensitivity and the guessed fault sensitivity. The correct key is the best match between these two [3].

6.3 Data Dependency of Fault Sensitivity on S-box Architectures

One of the most important requirements of FSA is that the attacker must be aware of the data dependencies that enable fault sensitivity. This section analyzes two different S-box implementations. The first one is the PPRM1 S-box, as studied in Li *et al.* [3]. The second architecture is the Boyar-Peralta S-box, a design with a very small gate footprint. The data presented in this section was extracted from an actual FPGA prototype, which will be discussed in a later section (Section VI).

6.3.1 PPRM1 S-box

Figure 6.2 shows the relation between the Hamming Weights of the inputs vs. the critical timing delay of the circuit. The initial values of all S-box architectures are set to zero. The input sequence is exhaustive, and assigns every possible input value. As shown in the graph,

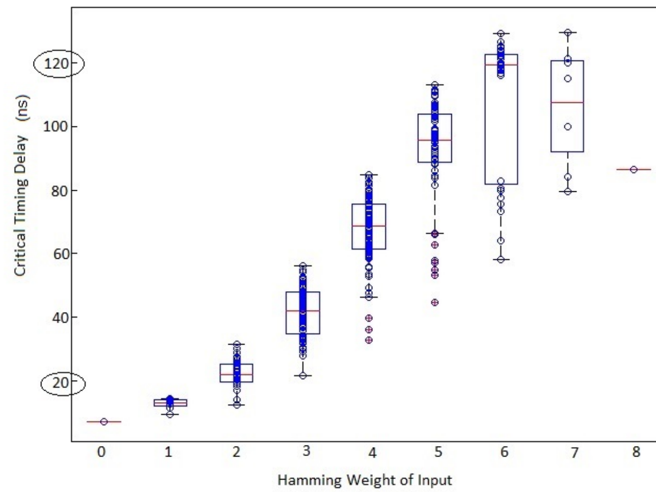


Figure 6.2: Data Dependency of Fault Sensitivity with Input Hamming Weight in PPRM1 S-box (Note the Y-scale Range)

the higher the Hamming Weight the larger the critical timing delay. This way, the attacker can extract the data dependency of fault sensitivity by different input values.

6.3.2 Boyar-Peralta S-box

Circuit minimization for the AES S-box is a widely studied hardware problem. A recent effort in this area is by Boyar and Peralta [56]. This design minimizes the total number of gates and the overall circuit depth. This design is a two step process. The first one is the non-linear gate reduction which is based on performing multiplication and inverse operations in large fields by implementing them in smaller fields[57]. We study this S-box because of its compactness, which makes it a good candidate for cost-sensitive, embedded cryptographic

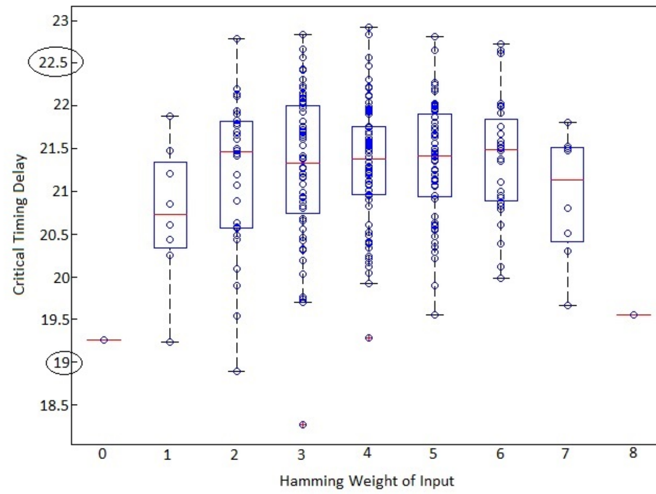


Figure 6.3: Data Dependency of Fault Sensitivity with Input Hamming Weight in Boyar-Peralta Sbox (Note the Y-scale Range)

applications.

We measured the data dependency of fault sensitivity in Boyar-Peralta's S-box. Figure 6.3 shows the critical timing delay of the S-box vs. the Hamming Weight of the input data. As shown, the range of mean critical timing delay in Figure 6.3 is between 19.5 to 21.5 while this range is from 10 to 120 in Figure 6.2. Consequently, the graph in Figure 6.3 discloses significantly less data dependency on fault sensitivity, when compared to Figure 6.2.

We also did not find any significant data dependencies of fault sensitivity using the factors such as:

- Use of a linear weighted combination of the input bits, rather than the sum (Hamming Weight)

- Value of the S-box outputs
- Hamming Distance among successive inputs
- The linear combination of edge triggers of input bits along with their values

The above experiments have been done on Canright S-box as well [35]. Based on the results, Canright S-box is more vulnerable to FSA attack than Boyar-Peralta S-box. The reason will be discussed in section IV.

6.4 Data Analysis

As shown in Figure 6.3, the range of the delays for different input Hamming weights is distributed uniformly in the case of the Boyar Peralta S-Box. However, this distribution for the PPRM1 S-box is based and depends on the Hamming weight value. A careful comparison of both architectures leads to the following observations.

- The AND network is the determining factor in data dependency of fault sensitivity. Figure 6.4 shows that the AND network for PPRM1 has AND arrays of depth 6 down to depth 1. However, as stated in Figure 6.5, the AND network for Boyar-Peralta S-box has only arrays of depth 2 and 1. Based on our analysis, the depth of the AND network in a circuit is an important factor to determine its sensitivity to setup time violations. If there are several AND networks in a circuit with different depths, and the difference between their depth is large, the data dependency of fault sensitivity would

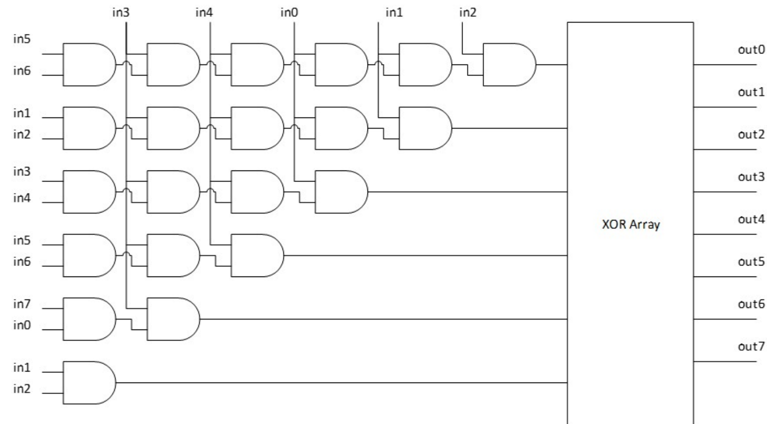


Figure 6.4: Partial Structure of the PPRM1 S-box Design

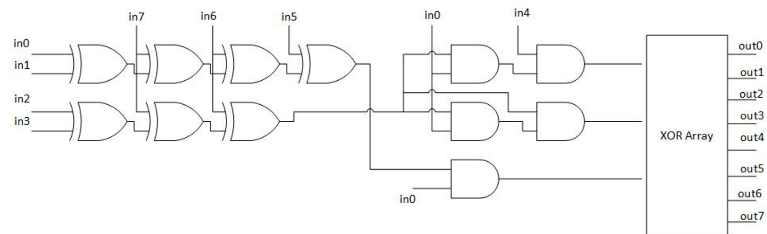


Figure 6.5: Partial Structure of Boyar-Peralta S-box Design

increase. We know that PPRM1 S-box has AND network of depth 7 down to depth 1. If we assume that each AND gate has a delay equal to T_{AND}, then the range of timing delay for PPRM1 S-box would be from $7 \times T_{AND}$ down to T_{AND}. This number for Boyar-Peralta would be $2 \times T_{AND}$ down to T_{AND} because it has AND network of depth 2 and 1. The delay difference for Boyar-Peralta is negligible compared to PPRM1.

- Based on Figure 6.4, the architecture of the PPRM1 S-box has a layer of AND gates and after that a layer of XOR gates. The only gates that affect the fault sensitivity are the AND and OR gates. So, in case of the PPRM1 S-box only a layer of the AND network

is counted for data dependency of fault sensitivity. However, for the Boyar-Peralta S-box (Figure 6.5), there is a layer of XOR gates before the AND network. The XOR network does not directly affect the data dependency of fault sensitivity, but the depth of the XOR network will affect the arrival time of the signals to the AND network. This increases the spread of the signals in time, *independent* of the actual data value. The net effect is that the contribution of the data-dependent delay of the AND network will decrease in relative terms. Hence, the XOR network preceding the AND network, in the case of the Boyar-Peralta S-box, will further decrease the fault sensitivity.

Based on the above discussion, the factors that affect a circuit to show data dependency of fault sensitivity are as follows:

1. The type of gates in the design: As observed in [3], the gates that cause data dependency of fault sensitivity are AND, OR and related combinations. XOR and XNOR gates do not affect the data dependency of the critical timing delay since their output always depends on both data inputs.

2. The differential depth of the effective gate network: If G_E is defined as the number of logic levels along any path from input to output that contain effective gates such as AND and OR, then the differential depth of the effective gate network is $\max G_E - \min G_E$,.

3. The arrival time of signals to the inputs of the effective gate network.

Based on the above discussion, the reason that Canright S-box is more vulnerable to FSA attack than Boyar-Peralta S-box is that Boyar-Peralta S-box is minimized for circuit depth. Meaning that the differential depth of effective gates is smaller in Boyar-Peralta S-box than Canright S-box. So, it is more difficult for the adversary to extract data dependency of fault sensitivity for Boyar-Peralta S-box.

6.5 Proposed Countermeasure

We propose a systematic countermeasure against the FSA attack. Finding the data dependency of fault sensitivity is the main issue for the attackers in FSA attack. Therefore, the proposed countermeasure is a masking method that aims at masking the factors that affect the data dependency of fault sensitivity. The main idea behind this countermeasure is to remove the dependency of the critical timing delay to the processed data values in the circuits. We propose a transformation that operates at two levels of abstraction, at netlist level and at gate level.

- Netlist level: The delay of the netlist must be independent of the input data.
- Gate level: The switching time of gates must be random during circuit evaluation, meaning that the switching distribution is uniform over the computation time of the circuit.

The proposed countermeasure is based on inserting delay elements in different paths of the circuit based on the statistical timing analysis of the circuit. The goal is to equalize the effective delay of each path in a circuit. The **Effective Delay of a Path** is defined as the number of effective gates in that path multiplied by their propagation delays. We start from the outputs of the circuit. For each output, we evaluate the effective delay to any input. We then find the maximum effective delay. We then insert delay elements near the input of each path such that the sum of effective path delay and inserted buffer delay becomes equal to the maximum effective delay. The number of delay elements is in inverse proportion to the length of the path.

6.5.1 FSA Resistant PPRM1 design

The PPRM1 S-box has been chosen as a case study. The delay elements are inserted in the path of each effective gate (AND gates) output until the delay of all of them becomes equal to the maximum delay of the circuit.

This FSA resistant design has been analyzed to find the data dependency. As shown in Figure 6.6, the timing delay is now uniform for each Hamming weight of the input values. The performance cost of this method is less than 1% since the final output would be ready at the maximum time in both FSA resistant and the original design. However, the number of gates has increased by 24%.

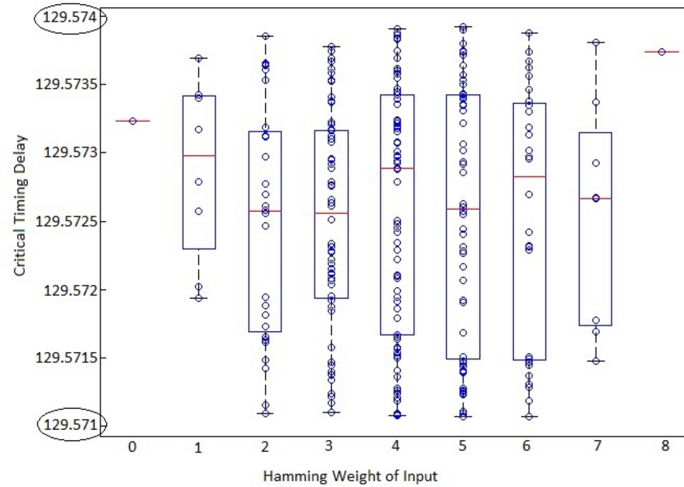


Figure 6.6: Timing Delays for FSA Resistant PPRM1 Design

6.5.2 FSA Resistant AES

In order to show the generality of the proposed countermeasure, we have implemented this countermeasure in a full round of Advanced Encryption Standard algorithm. AES is a symmetric key algorithm, the same key is used for both encryption and decryption [58]. The 128 bit key size AES, performs 10 rounds on the input block to generate the encrypted output, the so called ciphertext. Each round performs four operations, namely, Sub-Bytes, Shift-Rows, Mix-Columns and Add-Round-Key. We have studied the gate-level design for each of these operations to find the data dependency of fault sensitivity in their designs. The Mix-Columns algorithm shows the data dependency of fault sensitivity. The delay insertion algorithm has been applied to Mix-Columns as well. The final timing analysis of a round in AES algorithm is shown in Figure 6.7. The input sequence is a random number with the Hamming Weight between 0 and 128.

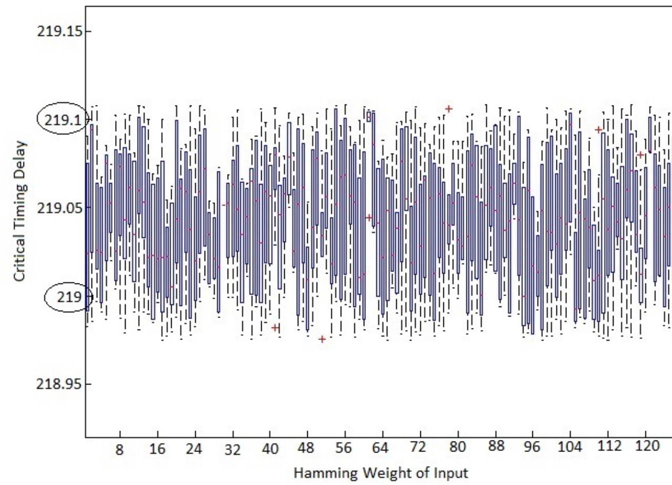


Figure 6.7: Timing Delays for FSA Resistant AES Design for one Round

6.6 Experimental Setup

This section clarifies the platform for calculation of Figure 6.3, 6.2, 6.6 and 6.7. Rather than using simulation tools, we created a test setup that enables us to determine the fault sensitivity of S-Box as implemented on an FPGA circuit. To make our results consistent with what we would expect from an ASIC design, we first mapped the S-Boxes under test to a gate-level netlist using the Synopsis Design Compiler [59] under the generic technology library. Next, the resulting netlist was translated in Verilog, constrained for FPGA synthesis, and integrated into our FPGA-based measurement circuit. We used the *Altera DE0-Nano* FPGA board for our implementations.

Figure 6.8 shows the hardware architecture of on-chip time measurement for DUT operations. The DUT can be any combinational logic. In this chapter we have replaced it with different

S-box implementations and a whole round of AES. The architecture consists of two blocks: a Trigger Circuit and a Measurement Circuit. The Trigger Circuit is used to generate an unstable (oscillating) feedback loop. The Measurement Circuit evaluates the period of this oscillation by comparison with a reference clock.

The feedback loop of Trigger Circuit is designed using only combinational logic without any pipeline stages. The combinational path consists of an *S-box* and decision/selection logic. The timing of this path is dominated by the DUT operation. Two equality checkers determine when the DUT output is generated. Then, the decision is captured in an *S-R latch* to prevent glitches. If enabled, the output of the *S-R latch* choose the next input. If *reg_0*, *reg_1*, *reg_2* and *reg_3* is set accordingly, the input of the DUT switches between *reg_0* and *reg_1*. The Measurement Circuit calculates the oscillation period. The select signal of DUT input multiplexer is also tied to the clock input of a counter. At every rising clock edge the value of the *cnt* register is incremented by 1. A reference counter (*ref_cnt*) using the system clock keeps the track of a known time. Then, we can calculate the oscillation time by $(\frac{ref_cnt}{cnt}) \times clk_period$ where *clk_period* is the clock period of the signal *system_clk*.

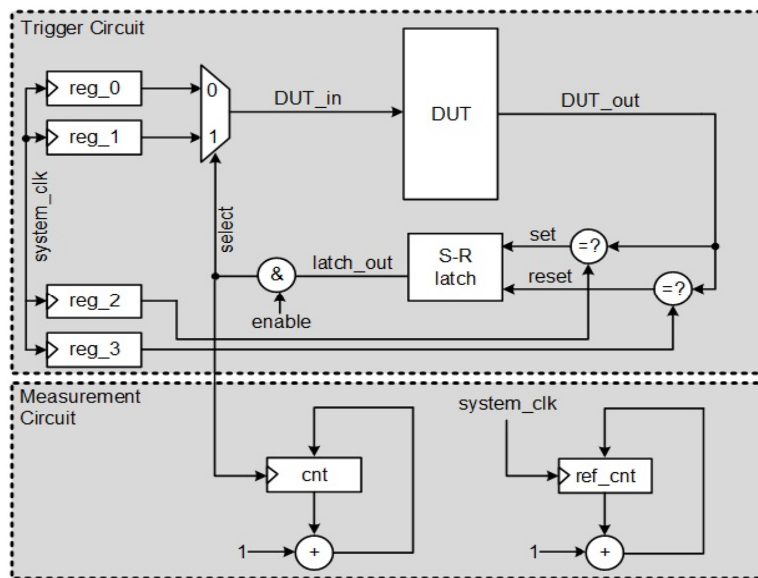


Figure 6.8: The architecture of the Trigger and Measurement Circuit for on-chip time measurements of S-box operations

Chapter 7

A Design Metric to Quantify Fault Attack Sensitivity

This paper is the first to propose an evaluation metric for assessing the vulnerability of a secure hardware design to fault attacks. This metric considers both generating an exploitable fault and observing the effect of the fault. We divide a cryptosystem into two parts. SoV covers the timing and location of the fault. SoP covers the fault propagation to the output. We perform STA on the SoV to find the likelihood of the fault generation. Then we perform the observability analysis to quantify the propagation of the fault to the output. We also show a case study of this factor on AES algorithm, which compares two fault attacks and two AES implementations.

7.1 A Design Metric for Fault Attack Evaluation

Fault attacks pose a serious threat to the security of cryptosystems [60]. A design metric to assist cryptographic hardware designers in the evaluation of the fault attack threat is vital because of the huge pool of fault attacks and possible countermeasures. However, the bulk of the research on fault attacks makes abstraction of the implementation, and concentrates instead on the analysis of fault effects in cryptographic algorithms [61]. Furthermore, the research efforts that consider actual fault attack implementations typically ignore their generalization as a design method or metric [62]. In this chapter, we investigate the feasibility of a metric that can be computed from a circuit's netlist and express the circuit's vulnerability to fault attacks.

Although each fault attack has different strategies and requirements, all of them share three requirements (Fig. 7.1). First, an attacker needs to inject an exploitable fault that results in the fault model (i.e., expected fault behavior of the attacked device). For this purpose, the attacker may use a variety of fault injection techniques such as overclocking and voltage-starving. Second, the attacker needs to propagate the effect of the injected fault to the output of the device. Third, the attacker needs to analyze the observed faults to find the secret key. An attacker can use several fault analysis methods (i.e., fault attack strategy) to exploit an observed fault such as Differential Fault Analysis [24], Fault Sensitivity Analysis [3], and Differential Fault Intensity analysis [44]. A successful attack is the one that meets these requirements.

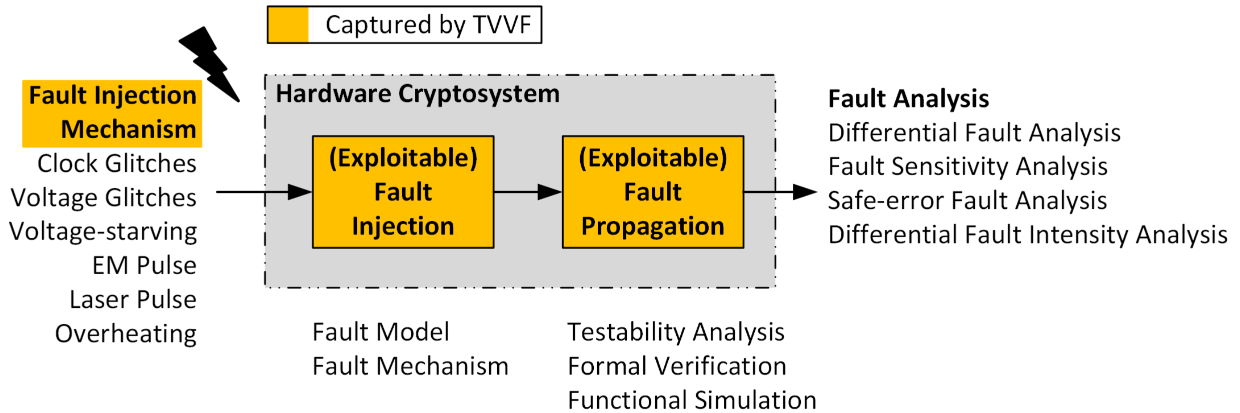


Figure 7.1: Requirements of a successful fault attack

To our knowledge, this is the first work to propose a metric for measuring the vulnerability of an architecture to fault attacks, the so-called **Timing Violation Vulnerability Factor** (TVVF). Given a hardware structure and a fault attack strategy, the TVVF quantifies the vulnerability of a circuit as a number between 0 and 1, representing non-vulnerable and completely vulnerable, respectively. TVVF estimates the feasibility of an attack in practice. Thus, it captures two crucial requirements of a fault attack.

- *How easy is it to inject an exploitable fault?*

  The likelihood of injecting an exploitable fault depends on the fault injection mechanism and fault model. This work focuses on the fault mechanisms that are based on the violation of setup timing constraints of a device. Therefore, our method relies on Static Timing Analysis (STA) to estimate the likelihood of injecting an exploitable fault.

  Timing violations can be obtained through different fault injection methods, including

overclocking, voltage-starving, and heating of a device [31]. We note that there are also fault injection mechanisms that generate single-event upsets, such as when using lasers or EM pulses. For these mechanisms, other methods than STA can be used for the likelihood of injecting an exploitable fault.

- *How easy is it to propagate the effect of the exploitable faults to the output?*

  We use the observability analysis to find the probability that an exploitable fault affects the device output. We use the observability analysis as the proof-of-concept method because of its simplicity. More advanced methods (e.g., formal verification) can be used for more accurate results.

The proposed methodology for computing TVVF does not require test vectors and can be a part of the design phase. The only assumption in this method is that the evaluator is in possession of the netlist of the hardware structure. TVVF concentrates on timing faults. However, our method for computing TVVF can be applied to any fault injection mechanism by using an appropriate method for likelihood of exploitable fault injection for this specific fault mechanism.

7.2 Timing Violation Vulnerability Factor

Timing Violation Vulnerability Factor (TVVF) of a hardware structure can be defined as the probability of injecting an exploitable fault and observing its effect in the output. We provide

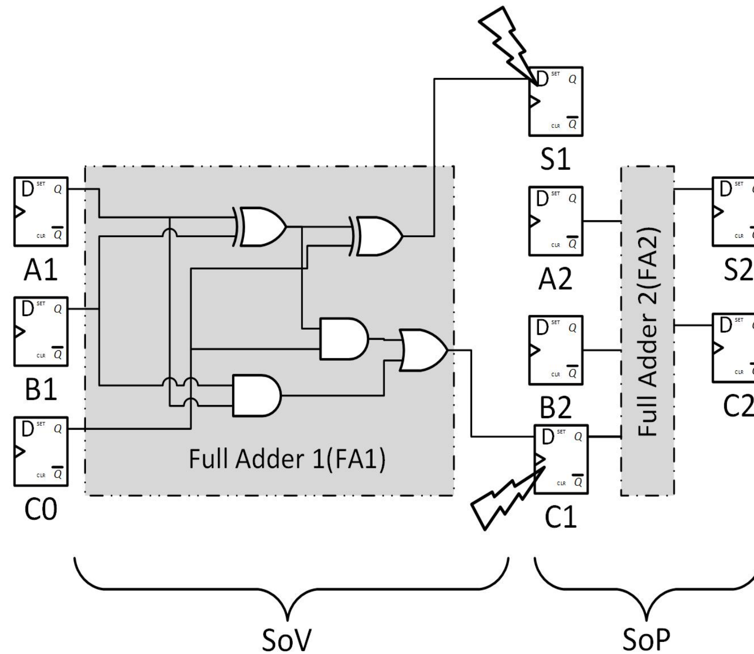


Figure 7.2: 2-bit Ripple Carry Adder (RCA)

a method to compute TVVF of hardware structures for a specific group of fault attacks, which use violation of timing constraints as the fault injection mechanism. Throughout the paper, we use overclocking (i.e., clock glitching) as the fault injection means for simplicity.

To define the concepts in a more convenient way, we use a 2-bit Ripple Carry Adder (RCA) as an example hardware structure (Fig. 7.2). In this example, our hypothetical attack strategy is to inject a random fault into the output of Full Adder 1 ($C1$ and $S1$) and observe the fault in the output of the RCA ($C2$ or $S2$ or $S1$).

The first requirement of a fault attack is injecting an exploitable fault. In a hardware design, there are multiple components that may contribute to a given fault. In a Setup Timing Violation attack on a register, for example, the entire logic cone at the data input

can potentially affect the Setup Timing. We use the term **Scope of Vulnerability** (SoV) to describe the collection of hardware elements that contribute to an exploitable fault. Referring to the example in Fig. 7.2, the SoV for faults on $C1$ or $S1$ is the first full adder($FA1$).

The second requirement is the ability to observe the exploitable fault. We use the term **Scope of Propagation** (SoP) to describe the part of a netlist that contributes to the propagation of such an exploitable fault. Referring to the RCA example, the SoP for faults on $C1$ or $S1$ is the second full adder, and we observe the effect of these faults through $C2$, $S2$, or $S1$. The SoP and SoV concepts apply equally well to cryptographic algorithms. For example, a given fault attack on AES requires the adversary to inject a fault into a state byte in the 9th round [63]. The SoV for this attack is SubBytes of 9th round and the SoP is the path from input of ShiftRows at the 9th round to the output of 10th round.

The concept of TVVF can be used for (i) comparing the vulnerability of two hardware implementations to a given fault attack and (ii) comparing the feasibility of two fault attacks on a specific hardware implementation. Therefore, each attack strategy A on an implementation C has a value for $TVVF(C, A)$. Each $TVVF(C, A)$ has an SoV and SoP component. In the next three subsections, we present a method to compute TVVF. The method includes analysis of SoV, analysis of SoP, and combining them to TVVF.

7.2.1 Timing characterization of the SoV

A timing violation attack is based on making a circuit fail its timing constraints. The attacker violates the timing constraints by controlling the clock period. For example, the injection of a glitch shortens the clock period. A fault manifests itself as a bit-flip on a violated path [62]. Hence, for a given clock period, the occurring fault type is the number of bit-flips at the SoV output.

We want to evaluate the likelihood of a bit-flip fault for a given clock period. We do this by analyzing the timing of each path in the circuit, and evaluating how many paths violate a given clock period constraint. We assume that the likelihood of a bit-flip fault is proportional to the number of violated paths. To characterize the circuit, we repeat this analysis for different clock period values. For each clock period, we count the number of violated paths. This count is used in the computation of bit-flip fault probability for each clock period, which is described in Algorithm 3. Table 7.1 shows the notations that we use throughout this paper.

In our RCA example, the attacker aims to inject a random fault into $FA1$. Therefore, we perform a timing characterization for $FA1$ using Algorithm 3:

- **Step 1:** We assume $T_{min} = 0.9ns$ and $T_{step} = 0.1ns$, as it can be seen in the first column of Table 7.2.
- **Step 2:** Using STA, we determine the critical path delay T_{max} of the RCA as $1.153ns$. Therefore, we need to apply three different clock periods ($G_{max} = \lceil(1.153 - 0.9)/0.1\rceil$

Table 7.1: Table of Notations

Notation	Definition
O_{max}	Total number of the SOV outputs
O_j	Output bit j of the SoV
p_{kj}	Timing path k to O_j
T_{kj}	Delay of p_{kj}
G_i	Clock period i
G_{max}	Total number of clock periods to characterize the SoV
P_{G_i,O_j}	Probability of injecting a bit-flip fault into O_j for a G_i
$P^n_{G_i}$	Probability of injecting n-bit-flip fault into the SoV for a G_i
P^n	Probability of injecting n-bit-flip fault into the SoV

= 3) to characterize the SoV (i.e, $FA1$).

- **Step 3:** After analysis of the STA results, we obtained the path delays for each RCA output. The T_{k1} and T_{k2} columns of Table 7.2 show these delays for $C1$ and $S1$, respectively.
- **Step 4:** We discard two paths annotated with $\times$ in Table 7.2. The delay values of these paths are smaller than T_{min}. Thus, the attacker is not able to violate these paths.
- **Step 5:** The columns N_{i1} and N_{i2} of Table 7.2 show the number of violated paths for each clock period. We compute the probability values P_{G_i,O_1} and P_{G_i,O_2} by using the values of N_{i1} and N_{i2}, respectively. The computed values are shown in Table 7.2.

Algorithm 3 P_{G_i,O_j}: Probability of injecting a bit-flip fault into an output O_j of SoV for a clock period G_i

1: // Specify the capabilities of the adversary

T_{min} = The smallest clock period that can be applied

T_{step} = The clock period resolution

2: // Perform Static Timing Analysis (STA) on SoV

T_{max} = The critical path delay of the circuit

$G_{max} = \lceil (T_{max} - T_{min})/T_{step} \rceil$

for each output O_j of SoV **do**

3: // Analyze STA results for each SoV output

$Nmax_j$ = Total number of timing paths to O_j

Analyze STA results to find the delay T_{kj} of each path p_{kj}

4: // Discard all paths that cannot be violated

for each path p_{kj} **do**

if $T_{kj} < T_{min}$ **then**

Discard path p_{kj}

end if

end for

5: // Compute P_{G_i,O_j} values

for each clock period G_i **do**

N_{ij} = The number of violated paths for clock period G_i

$P_{G_i,O_j} = N_{ij}/Nmax_j$

end for

end for

Table 7.2: Timing Characterization on $FA1$ for RCA example

Clock Period (G_i (ns))	Output 1 (C_1) $P_{obs}(C_1) = 0.5$			Output 2 (S_1) $P_{obs}(S_1) = 1$			$P^1_{G_i}$	$P^2_{G_i}$
	$T_{k1}(ns)$	N_{i1}	P_{G_i,O_1}	$T_{k2}(ns)$	N_{i2}	P_{G_i,O_2}		
$G_0 = 1.1$	1.029	0	0	1.153	1	0.33	0.33	0
$G_1 = 1.0$	1.029	1	0.33	1.153	1	0.33	0.261	0.054
$G_2 = 0.9$	0.968	2	0.66	0.968	2	0.66	0.261	0.217
-	0.834 ×	-	-	0.838 ×	-	-	-	-
Probability of Injecting n bit fault P_n							**0.284**	**0.09**

The probability of propagating an exploitable fault from output of the SoV to the output of the cryptosystem is determined by the circuit topology and processed data. Thus, any method that considers these factors can be used to compute this probability.

In this work, we use a probability-based observability analysis method for the probability of propagating an exploitable fault to the output. Observability analysis, which is widely used in VLSI test area, reflects the difficulty of propagating the value of a signal to primary outputs [21]. We selected this method as the proof-of-concept method due to its simplicity. More advanced methods can be used for better results.

In our method, we compute three probability values for each signal s of the SoP: $P_{s0}(s)$,

Table 7.3: Observability Computation Rules for Different Gates

Gate Type	$P_{s0}(out)$	$P_{s1}(out)$	$P_{obs}(in_i)$
AND	$1 - P_{s1}(out)$	$\prod_i P_{s1}(in_i)$	$P_{obs}(out) \times \prod_{i \neq j} P_{s1}(in_j)$
OR	$\prod_i P_{s0}(in_i)$	$1 - P_{s0}(out)$	$P_{obs}(out) \times \prod_{i \neq j} P_{s0}(in_j)$
XOR	$1 - P_{s1}(out)$	$P_{s1}(in_1)P_{s0}(in_0) + P_{s1}(in_0)P_{s0}(in_1)$	$P_{obs}(out) \times \max_{i \neq j}(P_{s0}(in_j), P_{s1}(in_j))$
Branch/Stem	$P_{s0}(stem)$	$P_{s1}(stem)$	$\max(Branch\ observabilities)$

$P_{s1}(s)$, and $P_{obs}(s)$. $P_{s0}(s)$ and $P_{s1}(s)$ are the probability of setting signal s to 0 and 1 by a random input vector, respectively. $P_{obs}(s)$ is the probability of observing the value of signal s at primary outputs (i.e., observability of s). We first compute $P_{s0}(s)$ and$P_{s1}(s)$ for each signal of the SoP in a breadth-first manner from inputs of the SoP (i.e., outputs of SoV) to the outputs of the SoP. After computing all $P_{s0}(s)$ and $P_{s1}(s)$ values, we compute the observability of each signal in a breadth-first manner from outputs of the SoP to inputs of the SoP. In these computations, $P_{s0}(s)$ and $P_{s1}(s)$ values of the SoP inputs are assigned to 0.5, and $P_{obs}(s)$ values of the SoP outputs are assigned to 1 as boundary conditions.

In Table 7.3, we provide computation rules for AND, OR, and XOR gates as an example. By applying the rules in Table 7.3, and starting from the given boundary condition, we can compute $P_{s0}(s)$, $P_{s1}(s)$, and $P_{obs}(s)$ values for any circuit consisting of these gates. For instance, we show $P_{s0}(s)$, $P_{s1}(s)$, and $P_{obs}(s)$ values of a full adder in Fig. 7.3. The three-value tuple v1/v2/v3 on each signal line corresponds to $P_{s0}(s)$, $P_{s1}(s)$, and $P_{obs}(s)$ values of this line, respectively.

For our RCA example, the SoP is the second full adder ($FA2$). SoV outputs are $S1$ and $C1$.

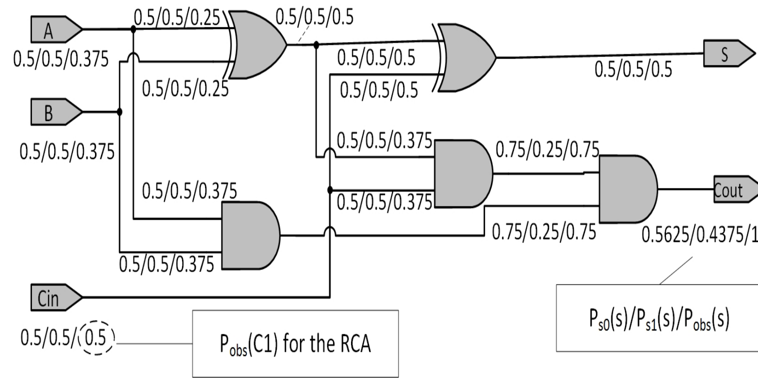


Fig. 7.3: Observability Analysis for a Full Adder Circuit

$P_{obs}(S1)$ is 1 because its is a primary output of the RCA. $P_{obs}(C1)$ is 0.5.

7.2.2 How do we compute TVVF for an attack on a circuit?

$TVVF(C, A)$ studies the probability of bit-flip faults in the outputs of a circuit C for a given attack model A. P^n is the probability of injecting n-bit-flips in the output of SoV and observing its effect in the output of SoP. Based on the required fault type for an attack, $TVVF(C, A)$ can be an appropriate P^n value or a combination of multiple P^n values. For example, we should select P^1 as the $TVVF(C, A)$ if our fault attack requires 1-bit fault. Similarly, we should select $P^1 + P^2 + ... + P^8$ if our fault attack requires a random byte fault. To compute TVVF, we need to compute all of the P^n values required by the fault model.

To observe an n-bit-flip in the output of SoV, we need n erroneous output bits. In other words, there are $\binom{O_{max}}{n}$ possible n-combinations of output bits that might yield an n-bit-flip. Here, O_{max} is the total number of SoV output bits. The contribution of each n-combination

to the P^n depends on three factors.

- *The probability of injecting a clock period* G_i: Since the attacker can inject at most G_{max} clock periods with the equal step size, this probability value is $\frac{1}{G_{max}}$.
- *The probability of obtaining a bit-flip in each output* O_j *of* SoV *for a clock period* G_i: Section III.A explains the method to compute P_{G_i,O_j}.
- *The probability of observing an exploitable fault in the output of the circuit*: Section III.B explains the method to compute $P_{obs}(O_j)$.

Algorithm 4 considers the contribution of each possible n-combination to compute P^n.

In our RCA example, we assume that the fault model requires random bit-flips at the output of $FA1$. To compute TVVF, we need to find P^1 and P^2 by using Algorithm 4. The results for P^1, P^2,$P^1_{G_i}$ and $P^2_{G_i}$ are shown in the last two columns of Table 7.2. In the following paragraph, we show the computation of $P^1_{G_2}$ and P^1 using Algorithm 4.

For computing $P^1_{G_2}$ we have two combinations. We can either inject fault in O_1 and observe it in the output, or inject fault in O_2 and observe it (Equation 7.1). To get P^1, we use Equation 7.2.

$$P^1_{G_2} = [P_{G_2,O_1} \times P_{obs}(O_1) \times (1 - P_{G_2,O_2})] + [(1 - P_{G_2,O_1}) \times P_{G_2,O_2} \times P_{obs}(O_2)] \quad (7.1)$$

Algorithm 4 P^n: Probability of obtaining n-bit fault at the SoV outputs and propagation of the fault to the SoP output

for each clock period G_i **do**

  for each n-combination $C \in \binom{O_{max}}{n}$ **do**

    $Temp = 1$

    for each output O_j of SoV **do**

      if $O_j \in C$ **then**

        $Temp \times = P_{G_i,O_j} \times P_{obs(O_j)}$

      else

        $Temp \times = (1 - P_{G_i,O_j})$

      end if

    end for

    $P^n_{G_i} += Temp$

  end for

  $P^n_{G_i} = P^n_{G_i}/G_{max}$

end for

$P^n = \sum_{\forall i} P_{G_i}$

$$P^1 = \frac{\sum_{i=1}^{3} P^1_{G_i}}{3} = 0.284 \tag{7.2}$$

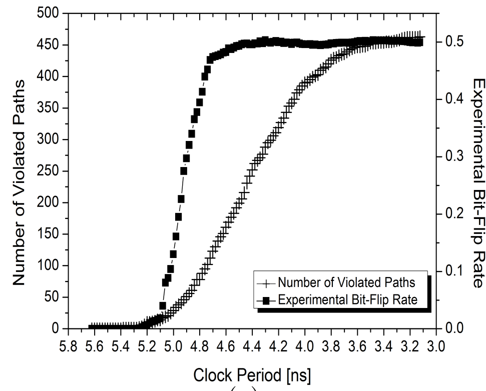
The final step of computing TVVF is combining appropriate P^n values. Since we require 2-bit random fault injection for our RCA example, we combine P^1 and P^2 to compute the TVVF as shown in Equation 7.3.

$$TVVF(RCA, 2\text{-}bit\text{-}flip) = P^1 + P^2 = 0.374 \tag{7.3}$$

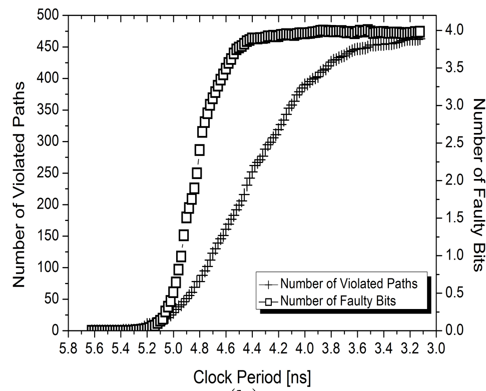
To compare multiple attacks on one implementation, we calculate the $TVVF(C, A)$ for each attack. Then the max($TVVF(C, A)$) value is selected as the TVVF of the hardware structure. The reason is that the TVVF must show the weakness of circuit against fault attacks. Therefore, we choose the highest probability (in other words the easiest attack on the implementation) as the TVVF of the circuit.

7.3 Results

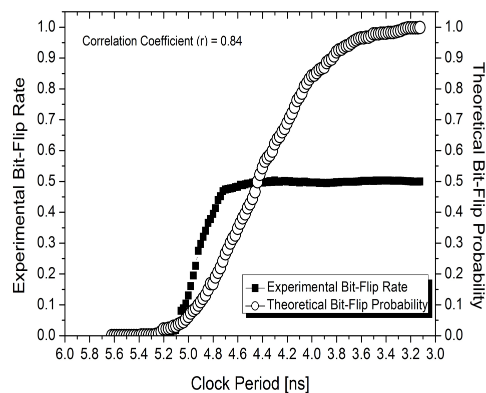
We performed two types of experiments. First (Section IV.A), we verified our base assumption that the probability of injecting an exploitable fault is proportional to the number of violated paths. Second (Section IV.B), we computed TVVF for two different attacks on two different AES hardware implementations.



(a)



(b)



(c)

Fig. 7.4: Experimental Verification of Fault Generation Probability for PPRM1 Sbox

7.3.1 Experimental Verification of Fault Generation Probability

In this subsection, we demonstrate that the probability of injecting an exploitable fault is proportional to the number of violated paths. We analyzed two AES SBox architectures: PPRM1 [33] and Boyar-Peralta [56]. We first generated Sbox netlists for an Altera Cyclone IV FPGA. Then, we applied gate-level timing simulation and STA.

First, we performed STA to obtain path delays. We applied Algorithm 3 and 4 to get the theoretical bit-flip injection probabilities. In the gate-level simulation, we gradually decreased the clock period from $6ns$ to $3.2ns$ with $20ps$ precision. For each step, we applied all possible 255^2 transitions to Sbox inputs and counted the number of bit-flips at each output bit. Then we computed the experimental bit-flip rate for each output bit, which is the ratio of the number of bit-flips to the total number of input transitions (i.e., 255^2). For each step, we also determined the number of violated paths within the fan-in cone of each output bit using the STA results.

Fig. 7.4(a) shows the experimental bit-flip rate and the number of violated paths for output bit 1 of the PPRM1 Sbox. The experimental bit-flip rate increases with the number of violated paths for a range of clock periods ($[5.621, 4.441] = 60 steps$). For smaller clock periods, the experimental bit-flip rate saturates around 0.5. This shows that after a certain clock period, the behavior of the circuit becomes random. After this point, the chance of getting a bit-flip in an output bit is 0.5. The other output bits of PPRM1 and Boyar-Peralta Sbox architectures also show similar behavior. Fig. 7.4(b) shows the average number of

bit-flips at the output of the PPRM1 Sbox and the number of violated paths. Again, the average number of faulty bits follows the number of violated paths for a certain range and then saturates at around 4. Hence, we can conclude that our assumption is valid.

As shown in Fig. 7.4(c), for bit 1 of the PPRM1 Sbox, there is a significant correlation (with a correlation coefficient of 0.84) between the experimental bit-flip rate and the theoretical bit-flip probability. However, the theoretical probability is optimistic compared to the experimental bit-flip rate. The reason behind this behavior is that we use simple assumptions in our computations. First, the probability computation in Algorithm 3 is a rough estimation for the actual fault behavior of the circuit. Second, we do not include a false-path analysis phase in STA, which makes our computations conservative. To address this problem, a false-path-aware STA method [64] can be used. Third, our computation assumes that every violated path contributes positively to the bit-flip probability. In practice, multiple faults that are caused by multiple violated paths may mask each other, and thus, they may negatively affect the bit-flip probability. This problem can be addressed by incorporating a functional simulation phase into the computation.

7.3.2 An Example Case Study

We applied our methodology for two attacks on two AES implementations as a case study. We computed TVVF for each possible *attack/implementation* pair. We first compared the feasibility of each attack on each implementation. Then, we compared the vulnerabilities of

the AES implementations to these attacks.

The aim of Piret attack [29] is to inject one random-byte fault into the output of SubBytes function of AES algorithm in round 9th. For Piret attack, the SoV is the Sbox of round 9th and the SoP is from the input of ShiftRows of round 9th to the output of round 10th. The Tunstall attack [61] relies on injecting one random-byte fault to the output of SubBytes of round 8th. For Tunstall attack, the SoV is the Sbox of round 8th and the SoP is from the input ShiftRows of round 8th to output of round 10th. For simplicity, we assume that the fault location is first state byte of the AES for each attack. Thus, SoV is the Sbox that corresponds to this byte.

Our AES netlists are generated for an FPGA. The only difference between these two AES implementations is their Sbox architectures. The first one, called *P-AES*, uses the PPRM1 [33] Sbox. The second one uses Boyar-Peralta [56] Sbox, which we call *B-AES*.

We first performed an STA analysis on both Sbox architectures. Then we included the observation probability in our computations. Table 7.4 shows the ratio of observability values for each attack. As it is shown, the observability of Piret attack is greater than of Tunstall attack. The reason is that Piret attack injects faults into round 9th and Tunstall attack injects faults to round 8th. The faulty values need to go through a shorter path in Piret compared to Tunstall attack. Then, we computed the TVVF for each attack using Algorithm 3 and 4. Table 4.3 shows the results of this experiment.

TVVF can be used to compare the feasibility of two attacks on a given hardware structure.

Table 7.4: $\frac{P_{obs}(Piret)}{P_{obs}(Tunstall)}$ for each output bit of SoV

$Piret/Tunstall$	O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8
PPRM1	27.2	51.1	132.6	8.2	37.3	24.9	58.1	59.7
Boyar-Peralta	42.5	67.4	34.1	49.6	49.6	27.5	67.4	34.5

Table 7.5: Comparison of TVVF for Two Fault Attacks on Two Implementations

Attack \ Circuit	*P-AES*	*B-AES*	$\frac{TVVF(P-AES)}{TVVF(B-AES)}$
Piret	2.81E-03	2.21E-03	1.27
Tunstall	5.26E-05	4.85E-05	1.08
$\frac{TVVF(Piret)}{TVVF(Tunstall)}$	53.4	45.5	–

As seen in Table 4.3, Tunstall attack is 53.4 and 45.5 times less feasible than the Piret attack on *P-AES* and *B-AES*, respectively. We can also use TVVF to compare the vulnerability of two implementations to a given attack. Table 4.3 shows that *P-AES* is 1.27 (1.08) times more vulnerable to Piret (Tunstall) attack than the *B-AES*. Due to the asymmetric structure of PPRM1 and delay-balanced structure of Boyar-Peralta, PPRM1 Sbox is more vulnerable to fault attacks than Boyar-Perlata [1].

7.4 Related Work

Many methods assess the reliability of a design to soft errors such as the Architectural Vulnerability Factor (AVF) [65]. These methods are widely used by fault tolerant system designers to make proper reliability/cost tradeoffs. However, they are not applicable to vulnerability analysis to fault attacks, because their assumption is random fault location and timing.

Feiten et. al [66] propose a methodology to compute the detection probability of a modeled fault. They model the faults with boolean satisfiability formulas and compute their detection probabilities using an automatic test pattern generation method. Their method only considers propagation probability of an exploitable fault, which corresponds to the SoP part of our method. They ignore the implementation aspects of the design and likelihood of injecting an exploitable fault, which is the SoV part of our method.

Chapter 8

Conclusion

Fault injection is a powerful hacking tool, affecting all forms of cryptography. As discussed in this dissertation, we describe common fault injection mechanisms, and common fault analysis techniques. The contributions of this thesis cover different aspects of fault attacks.

- This thesis presents a way to perform differential fault analysis based on the correlation of fault propagation probability and the value of faulty ciphertexts. The method requires reasonably more fault injections in comparison to previous works but has fewer restrictions on fault injection scheme. We have injected faults in the 10th round of AES algorithm and shown that by inversing the value of the faulty ciphertext byte to an intermediate value, namely the state byte of round 10, the location of the state bytes are close to each other only for the correct key byte. We have tested our methodology on an FPGA implementation of AES algorithm. Our results show that DFIA can

retrieve the 128-bit AES key by 7 fault injections. We also propose a DFIA on round-serial and nibble-serial implementations of PRESENT and LED. Based on our result, we can retrieve the unique key guess for each algorithm with a reasonable numb er of fault injections. Our method of fault injection is the clock glitching which is a very cheap and easy way of attacking for the adversary. We also propose an improvement on the DFIA attack that can recover the key using multiple plaintexts. We also study the relation between the number of plaintexts (encryptions) used, and the resolution of the fault injection equipment.

- In this thesis, we also presented a comparison of four recently published attacks that use fault bias, the non-uniform response of digital systems towards fault injection. By investigating the common elements of these attacks,we were able to build a single framework that supports a systematic comparison. Our main conclusion is that, even though all of the investigated attacks use the same test case of glitch injection on an AES design, their performance differs greatly. Using the number of fault injections as the cost metric, we found that DFIA performs best, followed by FSA, NUEVA and NUFVA. The main reason for the better performance of DFIA is the differential nature of the analysis mechanism. This makes DFIA more tolerant against estimation mistakes and noise effects. Also, DFIA can use the entire fault characteristic for a given input stimulus, in contrast to other fault injection te techniques, which uses only a single fault p er input stimulus. Overall, fault-bias based techniques are effective as an implementation attack, and they show that fault-injection based attacks can be

applied in a generic setting with minimal assumptions on the underlying cryptographic implementation. It seems reasonable to conclude that there is a rapidly increasing need to develop countermeasures against fault bias in digital hardware, including firmware-driven embedded systems.

- This thesis evaluates the cause of FSA by analyzing different S-box architectures. We have demonstrated the existence of two factors, the depth of the AND/OR network in a design, as well as the arrival time of input signals to the AND/OR network. Both of these factors influence the fault sensitivity. Based on these two factors, a countermeasure has been suggested to eliminate fault sensitivity in a design. The proposed method has been demonstrated in a prototype setup. In our current research, we are evaluating the automation of this design transformation to larger circuits.

- This thesis also proposes an evaluation metric for computing the vulnerability of a secure hardware design against fault attacks. In this metric, the user has to consider two important facts about a fault attack: generation of the fault model and observation of fault in the ciphertext. Therefore, we divide the cryptosystem into two parts. The first one is Scope of Vulnerability which defines the timing and location of the fault. Second is the Scope of Propagation which covers the fault location to the output of cryptosystem. We perform STA on the SoV to find the easiness of the fault generation. Then we perform the observability analysis which covers the propagation of the fault to the output. We also show a case study of this factor on AES algorithm and compare two different types of attack on two different hardware implementations of AES.

From these observations, we derive a set of guidelines and techniques for fault-attack resistant design. The main objective of this contribution is to describe fault-attack resistant design and differentiate it from fault tolerant design, a set of techniques based on redundancy. The key differentiator between the two types of design can be made by considering the cause of the fault. Fault tolerant design deals with random, arbitrary events and generic failures of a design. In contrast, fault-attack resistant design deals with an intelligent adversary who has a focused objective to break the security of a design. The fault tolerant methods basically require the system to be able to continue performing its functions correctly in presence of faults. On the other hand, a fault-attack resistant design requires the system to continue performing its intended operation without leaking secret-data dependent information in presence of faults. While fault tolerant design techniques can be used to create a fault-attack resistant design, in this chapter will show that by analyzing the fault attack requirements, the nature of the threat enables significant optimizations, which improve cost and performance of the protected designs. We review several fault-resistant design techniques that are generic and broadly applicable to secure intellectual property (IP) modules. The pyramid of fault measurement can be used by the designer to build a fault-attack resistant design considering the costs and security coverage of countermeasures in each step. The designer can find the optimized combination of countermeasures to prevent fault measurement for an attacker. While the pyramid provides a road map for designers to apply fault attack countermeasures, it does not provide information on cost and security efficiency of each countermeasure.

8.1 Overall Conclusion and Future Work

In this dissertation, we proposed novel and more powerful fault attacks, countermeasures and evaluation metrics for measuring the resistancy of a design against fault attacks. The observations and results of this dissertation lead to many future work including building a Fault-Attack Aware Microprocessor. Towards this end, my colleagues and I are working on several projects explained below:

- The first goal is to have better and deeper understanding of the faulty behavior of the microprocessors. To obtain this goal, we are performing a deep analysis of the pipeline behavior of the LEON3 microprocessor under gradual increase of fault intensity. We then, successfully, launch the DFIA attack on the LEON3 microprocessor. Our observations and analysis show that using our technique and using the knowledge of micro-architectural characteristics will help improve the efficiency of the attack by orders of magnitude. These analysis are also useful in other applications. An important and vital example of these applications is the reliability of the traditional software based countermeasures. Using our analysis, we are able to break the traditional software based countermeasures such as Instruction Duplication, Instruction triplication, Parity based countermeasures, etc.
- Using the results of the previous item, we come to the conclusion that a powerful and robust countermeasure against fault attack cannot be based only in software or hardware. Therefore, we propose FAME (Fault-Attack Awareness Using microproces-

sor Enhancements) processor. In this processor, by equipping the hardware with fast and low cost detection methods, we are able to detect the source of fault injection and anomalies in the processor behavior. Then, the critical data and the control of the processor is transfered to a secure interrupt routine to handle the policies after recognizing the fault attack.

- There are still several side projects that can be addressed as the future work of this thesis. Examples of these projects are launching attacks on light weight and public key cryptosystems such as Simon, RSA and Elliptic Curve Cryptographic (ECC) based algorithms. The DFIA attack can also be improved more to break the cryptosystems equipped with Side-Channel resistant countermeasures. This attack can also use other sources of leakage such as power consumption or number of transitions as a result of fault injection to converge to the correct key more efficient and faster.

The outcome of this dissertation has been published in 1 journal paper publication and 1 journal paper Submission, 1 book chapter, and 10 conference publications. This research has been recognized as the best paper in session by the SRC Techcon community at 2015. Two publications of this thesis are also awarded the best presentation and the best poster in CESCA Day 2014 and 2015 respectively. The author of this thesis has been recognized as the outstanding student in CESCA Day 2016.

Bibliography

[1] Farhady Ghalaty, N., Aysu, A., Schaumont, P.: Analyzing and eliminating the causes of fault sensitivity analysis. In: Proc. of Design, Automation and Test in Europe Conference and Exhibition (DATE). (2014) 1–6

[2] Ghalaty, N.F., Yuce, B., Taha, M., Schaumont, P.: Differential Fault Intensity Analysis. In: Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014 Workshop on, IEEE (2014) 49–58

[3] Li, Y., Sakiyama, K., Gomisawa, S., Fukunaga, T., Takahashi, J., Ohta, K.: Fault sensitivity analysis. In: Proc. of Cryptographic Hardware and Embedded Systems (CHES). (2010) 320–334

[4] Joye, M., Tunstall, M., eds.: Fault Analysis in Cryptography. Information Security and Cryptography. Springer (2012)

[5] Karaklajic, D., Schmidt, J.M., Verbauwhede, I.: Hardware Designer's Guide to Fault Attacks. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on **21** (2013) 2295–2306

[6] Barenghi, A., Bertoni, G., Parrinello, E., Pelosi, G.: Low voltage fault attacks on the rsa cryptosystem. In: Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on, IEEE (2009) 23–31

[7] Hutter, M., Schmidt, J.: The Temperature Side Channel and Heating Fault Attacks. In: Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers. (2013) 219–235

[8] van Woudenberg, J., Witteman, M., Menarini, F.: Practical Optical Fault Injection on Secure Microcontrollers. In: IEEE Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC),. (2011) 91–99

[9] Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. Proceedings of the IEEE **94** (2006) 370–382

[10] Barenghi, A., Breveglieri, L., Koren, I., Naccache, D.: Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures. Proceedings of the IEEE **100** (2012) 3056–3076

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

[12] Takahashi, J., Fukunaga, T., Gomisawa, S., Li, Y., Sakiyama, K., Ohta, K.: Fault Injection and Key Retrieval Experiments on an Evaluation Board. In Joye, M., Tunstall,

M., eds.: Fault Analysis in Cryptography. Information Security and Cryptography. Springer Berlin Heidelberg (2012) 313–331

[13] Balasch, J., Gierlichs, B., Verbauwhede, I.: An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs. In: Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on. (2011) 105–114

[14] Barenghi, A., Bertoni, G.M., Breveglieri, L., Pellicciolі, M., Pelosi, G.: Injection Technologies for Fault Attacks on Microprocessors. In Joye, M., Tunstall, M., eds.: Fault Analysis in Cryptography. Information Security and Cryptography. Springer Berlin Heidelberg (2012) 275–293

[15] Quisquater, J., Samyde, D.: Eddy Current for Magnetic Analysis with Active Sensor. In: Esmart. (2002)

[16] van Woudenberg, J., Witteman, M., Menarini, F.: Practical Optical Fault Injection on Secure Microcontrollers. In: Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on. (2011) 91–99

[17] Skorobogatov, S., Woods, C.: Breakthrough Silicon Scanning Discovers Backdoor in Military Chip. In: CHES. (2012) 23–40

[18] Skorobogatov, S.P.: Semi-invasive Attacks – A New Approach to Hardware Security Analysis. Technical Report UCAM-CL-TR-630, University of Cambridge, Computer Laboratory (2005)

[19] Skorobogatov, S.P., Anderson, R.J.: Optical Fault Induction Attacks. In: Cryptographic Hardware and Embedded Systems-CHES 2002. Springer (2003) 2–12

[20] Yuce, B., Ghalaty, N.F., Schaumont, P.: 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) 72–77

[21] Wang, L.T., Wu, C.W., Wen, X.: VLSI Test Principles and Architectures: Design for Testability. Academic Press (2006)

[22] Agoyan, M., Dutertre, J.M., Naccache, D., Robisson, B., Tria, A.: When Clocks Fail: On Critical Paths and Clock Faults. In: Smart Card Research and Advanced Application. Springer (2010) 182–193

[23] Boneh, D., DeMillo, R.A., Lipton, R.J.: On the Importance of Eliminating Errors in Cryptographic Computations. Journal of cryptology **14** (2001) 101–119

[24] Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Proc. of Advances in Cryptology (CRYPTO). (1997) 513–525

[25] Blömer, J., Seifert, J.P.: Fault based cryptanalysis of the advanced encryption standard (AES). In: Financial Cryptography, Springer (2003) 162–181

[26] Quisquater, J.J., Samyde, D.: Electromagnetic Analysis (EMA): Measures and Countermeasures for Smart Cards. In: Smart Card Programming and Security. Springer (2001) 200–210

[27] Moradi, A., Shalmani, M.T.M., Salmasizadeh, M.: A generalized method of differential fault attack against aes cryptosystem. In: Cryptographic Hardware and Embedded Systems (CHES). Springer (2006) 91–100

[28] Dehbaoui, A., Dutertre, J.M., Robisson, B., Orsatelli, P., Maurine, P., Tria, A.: Injection of Transient Faults using Electromagnetic Pulses-Practical Results on a Cryptographic System. IACR Cryptology ePrint Archive **2012** (2012) 123

[29] Piret, G., Quisquater, J.J.: A differential fault attack technique against SPN structures, with application to the AES and KHAZAD. In: Proc. of Cryptographic Hardware and Embedded Systems (CHES). Springer (2003) 77–88

[30] Clavier, C.: Attacking block ciphers. In Joye, M., Tunstall, M., eds.: Fault Analysis in Cryptography. Information Security and Cryptography. Springer Berlin Heidelberg (2012) 19–35

[31] Zussa, L., Dutertre, J.M., Clédiere, J., Robisson, B., Tria, A., et al.: Investigation of timing constraints violation as a fault injection means. In: Proc. of Design of Circuits and Integrated Systems (DCIS). (2012)

[32] Lashermes, R., Reymond, G., Dutertre, J., Fournier, J., Robisson, B., Tria, A.: A dfa on aes based on the entropy of error distributions. (2012) 34–43

[33] Morioka, S., Satoh, A.: An optimized s-box circuit architecture for low power AES design. In: Proc. of Cryptographic Hardware and Embedded Systems (CHES). (2003) 172–186

[34] Boyar, J., Peralta, R.: A New Combinational Logic Minimization Technique with Applications to Cryptology. In: Experimental Algorithms. Springer (2010) 178–189

[35] Canright, D.: A very compact s-box for aes. In: Cryptographic Hardware and Embedded Systems (CHES). Springer (2005) 441–455

[36] De Santis, F., Guillen, O., Sakic, E., Sigl, G.: Ciphertext-Only Fault Attacks on PRESENT. Third International Workshop on Lightweight Cryptography for Security and Privacy (2014) 84–105

[37] Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-lightweight Block Cipher. Springer (2007)

[38] Guo, J., Peyrin, T., Poschmann, A., Robshaw, M.: The LED Block Cipher. In: Cryptographic Hardware and Embedded Systems–CHES. Springer (2011) 326–341

[39] Altera Corporation: ModelSim Altera Starter Edition. (Available:www.altera.com)

[40] Jeong, K., Lee, C.: Differential Fault Analysis on Block Cipher LED-64. In: Future Information Technology, Application, and Service. Springer (2012) 747–755

[41] Endo, S., Sugawara, T., Homma, N., Aoki, T., Satoh, A.: An On-chip Glitchy-clock Generator for Testing Fault Injection Attacks. Journal of Cryptographic Engineering **1** (2011) 265–270

[42] Bagheri, N., Ebrahimpour, R., Ghaedi, N.: New Differential Fault Analysis on PRESENT. EURASIP Journal on Advances in Signal Processing **2013** (2013)

[43] Zhao, X.j., Guo, S., Zhang, F., Wang, T., Shi, Z., Ji, K.: Algebraic Differential Fault Attacks on LED Using a Single Fault Injection. IACR Cryptology ePrint Archive **2012** (2012) 347

[44] Farhady Ghalaty, N., Yuce, B., Taha, M., Schaumont, P.: Differential fault intensity analysis. In: Proc. of Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). (2014) 34–43

[45] Fuhr, T., Jaulmes, E., Lomné, V., Thillard, A.: Fault Attacks on AES with Faulty Ciphertexts Only. In: 2013 IEEE Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), IEEE (2013) 108–118

[46] Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A compact rijndael hardware architecture with s-box optimization. In: Advances in Cryptology (ASIACRYPT). Springer (2001) 239–254

[47] Li, Y., Hayashi, Y.i., Matsubara, A., Homma, N., Aoki, T., Ohta, K., Sakiyama, K.: Yet another fault-based leakage in non-uniform faulty ciphertexts. In: Foundations and Practice of Security. Springer (2014) 272–287

[48] Takahashi, J., Hayashi, Y.i., Homma, N., Fuji, H., Aoki, T.: Feasibility of Fault Analysis based on Intentional Electromagnetic Interference. In: Electromagnetic Compatibility (EMC), 2012 IEEE International Symposium on, IEEE (2012) 782–787

[49] Selmane, N., Bhasin, S., Guilley, S., Graba, T., Danger, J.L.: WDDL is protected against setup time violation attacks. In: Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on, IEEE (2009) 73–83

[50] Li, Y., Ohta, K., Sakiyama, K.: Revisit fault sensitivity analysis on WDDL-AES. In: Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on, IEEE (2011) 148–153

[51] Satoh, A., Sugawara, T., Homma, N., Aoki, T.: High-performance concurrent error detection scheme for aes hardware. In: Cryptographic Hardware and Embedded Systems (CHES). Springer (2008) 100–112

[52] Moradi, A., Mischke, O., Paar, C.: Collision timing attack when breaking 42 aes asic cores. Technical report, Cryptology ePrint Archive, Report 2011/162, 2011. http://eprint. iacr. org (2011)

[53] Moradi, A., Mischke, O., Paar, C., Li, Y., Ohta, K., Sakiyama, K.: On the power of fault sensitivity analysis and collision side-channel attacks in a combined setting. In: Cryptographic Hardware and Embedded Systems (CHES). Springer (2011) 292–311

[54] Endo, S., Li, Y., Homma, N., Sakiyama, K., Ohta, K., Aoki, T.: An efficient countermeasure against fault sensitivity analysis using configurable delay blocks. In: Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), IEEE (2012) 95–102

[55] Y., L., Sakiyama, K.: Toward effective countermeasures against an improved fault sensitivity analysis. IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences **95** (2012) 234–241

[56] Boyar, J., Peralta, R.: A small depth-16 circuit for the AES s-box. In: Proc. of Information Security and Privacy Research. (2012) 287–298

[57] Boyar, J., Peralta, R., Pochuev, D.: On the multiplicative complexity of boolean functions over the basis (and,xor,1). Theoretical Computer Science **235** (2000) 43–57

[58] Daemen, J., Rijmen, V., Proposal, A.: Rijndael. In: Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST). (1998)

[59] Compiler, D.: Synopsys inc (2000)

[60] Karaklajic, D., Schmidt, J.M., Verbauwhede, I.: Hardware designer's guide to fault attacks. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on **21** (2013) 2295–2306

[61] Tunstall, M., Mukhopadhyay, D., Ali, S.: Differential fault analysis of the advanced encryption standard using a single fault. In: Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication. Volume 6633 of Lecture Notes in Computer Science. Springer (2011) 224–233

[62] Bhasin, S., Selmane, N., Guilley, S., Danger, J.L.: Security evaluation of different AES implementations against practical setup time violation attacks in FPGAs. In: Proc. of IEEE International Symposium on Hardware-Oriented Security and Trust (HOST). (2009) 15–21

[63] Giraud, C.: DFA on AES. In: Advanced Encryption Standard–AES. Springer (2005) 27–41

[64] Liou, J.J., Krstic, A., Wang, L.C., Cheng, K.T.: False-path-aware statistical timing analysis and efficient path selection for delay testing and timing validation. In: Proc. of Design Automation Conference (DAC). (2002) 566–569

[65] Mukherjee, S., Weaver, C.T., Emer, J., Reinhardt, S.K., Austin, T.: Measuring architectural vulnerability factors. IEEE Micro **23** (2003) 70–75

[66] Feiten, L., Sauer, M., Schubert, T., Czutro, A., Bohl, E., Polian, I., Becker, B.: #SAT-based vulnerability analysis of security components a case study. In: Proc. of IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT). (2012) 49–54