Towards Comprehensive Side-channel Resistant Embedded Systems

Yuan Yao

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Engineering

Patrick R. Schaumont, Chair

Matthew Hicks

Leyla Nazhandali

Daphne Yao

Jeffrey H. Reed

July 27, 2021

Blacksburg, Virginia

Keywords: Side-Channel Attacks, Pre-silicon, Side-channel Leakage Source,

Countermeasures, Simulation, Root Cause Analysis, Leakage Evaluation

Copyright 2021, Yuan Yao

Towards Comprehensive Side-channel Resistant Embedded Systems

Yuan Yao

(ABSTRACT)

Side-channel leakage, which reveals the secret information from the physical effects of computing secret variables, has become a serious vulnerability in secure hardware and software implementations. In side-channel attacks, adversaries passively exploit variations such as power consumption, timing, and electromagnetic emission during the computation with secret variables to retrieve sensitive information. The side-channel attack poses a practical threat to embedded devices, an embedded device's cryptosystem without adequate protection against side-channel leakage can be easily broken by the side-channel attack.

In this dissertation, we investigate methodologies to build up comprehensive side-channel resistant embedded systems. However, this is challenging because of the complexity of the embedded system. First, an embedded system integrates a large number of components. Even if the designer can make sure that each component is protected within the system, the integration of the components will possibly introduce new vulnerabilities. Second, the existing side-channel leakage evaluation of embedded system design happens post-silicon and utilizes the measurement on the prototype of the taped-out chip. This is too late for mitigating the vulnerability in the design. Third, due to the complexity of the embedded system, even though the side-channel leakage is detected, it is very hard to precisely locate the root cause within the design. Existing side-channel attack countermeasures are very costly in terms of design overhead. Without a method that can precisely identify the side-channel leakage source within the design, huge overhead will be introduced by blindly add the side-channel countermeasure to the whole design. To make the challenge even harder, the Power Distribution Network (PDN) where the hardware design locates is also vulnerable to side-channel attacks. It has been continuously demonstrated by researchers that attackers

can place malicious circuits on a shared PDN with victim design and open the opportunities for the attackers to inject faults or monitoring power changes of the victim circuit.

In this dissertation, we address the challenges mentioned above in designing a side-channel-resistant embedded system. We categorize our contributions into three major aspects—first, we investigating the effects of integration of security components and developing corresponding countermeasures. We analyze the vulnerability in a widely used countermeasure - masking, and identify that the random number transfer procedure is a weak link in the integration which can be bypassed by the attacker. We further propose a lightweight protection scheme to protect function calls from instruction skip fault attacks. Second, we developed a novel analysis methodology for pre-silicon side-channel leakage evaluation and root cause analysis. The methodology we developed enables the designer to detect the side-channel leakage at the early pre-silicon design stage, locate the leakage source in the design precisely to the individual gate and apply highly targeted countermeasure with low overhead. Third, we developed a multipurpose on-chip side-channel and fault monitoring extension - Programmable Ring Oscillator (PRO), to further guarantee the security of PDN. PRO can provide on-chip side-channel resistance, power monitoring, and fault detection capabilities to the secure design. We show that PRO as application-independent integrated primitives can provide side-channel and fault countermeasure to the design at a low cost.

Towards Comprehensive Side-channel Resistant Embedded Systems

Yuan Yao

(GENERAL AUDIENCE ABSTRACT)

Embedded devices almost involve every part of our lives, such as health condition monitoring, communicating with other people, traveling, financial transactions, etc. Within the embedded devices, our private information is utilized, collected and stored. Cryptography is the security mechanism within the embedded devices for protecting this secret information. However, cryptography algorithms can still be analyzed and attacked by malicious adversaries to steal secret data. There are different categories of attacks towards embedded devices, and the side-channel attack is one of the powerful attacks. Unlike analyzing the vulnerabilities within the cryptography algorithm itself in traditional attacks, the side-channel attack observes the physical effect signals while the cryptography algorithm runs on the device. These physical effects include the power consumption of the devices, timing, electromagnetic radiations, etc., and we call these physical effects that carry secret information side-channel leakage. By statistically analyzing these side-channel leakages, an attacker can reconstruct the secret information.

The manifestation of side-channel leakage happens at the hardware level. Therefore, the designer has to ensure that the hardware design of the embedded system is secure against side-channel attacks. However, it is very arduous work. An embedded systems design including a large number of electronic components makes it very difficult to comprehensively capture every side-channel vulnerability, locate the root cause of the side-channel leakage, and efficiently fix the vulnerabilities. In this dissertation, we developed methodologies that can help designers detect and fix side-channel vulnerabilities within the embedded system design at low cost and early design stage.

Dedication

I dedicate this to my parents, Zhihong Zhang & Runya Yao,

my beloved soulmate Yuan Liu,

and my family.

I couldn't have made it this far without your selfless love, support and encouragement.

I specifically dedicate this to the loving memory of my grandparents. Grandma and grandpa, I hope you are proud of me there in heaven.

Acknowledgments

I would like to sincerely thank my advisor Dr. Patrick Schaumont, for his support, advice, and encouragement throughout my entire Ph.D journey. I still remember how naive in research I was when I first walked into his office five years ago. Through this journey, he teaches me lots of valuable things with patience on how to be an outstanding researcher as well as be a kind person. His perseverance, dedication, integrity and enthusiasm significantly influence my attitude towards life and career. He has set an example of excellence as a researcher and advisor. None of this work would be possible without his guidance and devotion to my research.

My gratitude also goes to my committee members Prof. Daphne Yao, Prof. Leyla Nazhandali, Prof. Matthew Hicks, Prof. Jeffrey Reed, for their valuable support and for offering helpful feedback on my research.

I would like to deeply appreciate the support from my co-authors and labmates. Thank you Tarun Kathuria, Mo Yang, Conor Patrick, Chinmay Deshpande, Daniel Dinu, Tuna Tufan, Marjan Ghodrati, Bilgiday Yuce, Abhishek Ajey Bendre, Richa Singh, Archanaa Santhana Krishnan, Pantea Kiaei. You let me feel like I have a nice lab family.

It's my great honor to have the opportunities to work with extremely talented researchers during my internships with Riscure B.V., Cryptography Research, Inc and Qualcomm. Thank you Dr. Baris Ege, Dr. Michael Tunstall, Dr. Elke De Mulder, Anton Kochepasov, Gilbert Goodwill, Dr. Nazanin Takbiri and Jon Azen. I have learned a lot from every one of you.

I would like to specifically thank my friends, Archanaa Santhana Krishnan and Pantea Kiaei. This has been a challenging time, thank you very much for your support and care through

hardness. I feel so lucky and so happy to have you being around in my life.

Thank you, Yuan Liu, you let me understand that the most long-lasting expression of love is accompanying. I never feel lonely and timid along with this journal because you always stand by my side.

Finally, special words go to my parents. I experienced a lot and went through a long journey to get here, you always stand behind me and give me unconditional love and support whenever I needed. You raise me up to more than I can be.

Contents

List of Figures **xv**

List of Tables **xxii**

1 Introduction **1**

1.1 Side-channel Analysis of Embedded Systems 4

1.1.1 Example of Side-channel Analysis 6

1.1.2 Existing Side-channel Countermeasures 9

1.2 Secure Embedded System Design Challenges 10

1.3 Contributions of this Dissertation and Outline 12

2 Fault-assisted Side-channel Analysis of Masked Implementations **17**

2.1 Introduction 17

2.2 Background 20

2.2.1 Attacker Model 20

2.2.2 Signal Processing Tools 20

2.3 Implementation of Masking Countermeasure 23

2.3.1 Pseudo Random Number Generation 23

2.3.2 Byte-level Masked AES Implementation 24

2.3.3 Disabling the Mask 25
2.4 Methodology 26
2.4.1 Analyze Mask Generation 26
2.4.2 Tuning Fault Injection Parameters 30
2.4.3 Fault Injection and Differential Power Analysis 31
2.5 Experimental Setup 32
2.6 Case Study 33
2.6.1 Bit-Sliced AES Implementation 33
2.6.2 Bit-Sliced masked AES Implementation 34
2.6.3 Steps of methodology 35
2.6.4 Results 38
2.7 Discussion 40
2.7.1 Fault Injection Success Rate 40
2.7.2 Related Work 40
2.7.3 Possible countermeasures 41
2.8 Conclusion 42

3 A Low-cost Function Call Protection Mechanism Against Instruction Skip Fault Attacks 43

3.1 Introduction 43
3.1.1 Motivation 44

3.1.2 Contributions 46
3.2 Background 47
3.2.1 Fault detection principles 47
3.2.2 Application Binary Interfaces 49
3.3 Countermeasures 51
3.3.1 Fundamental Principles 51
3.3.2 Methodology 54
3.3.3 ABI specific Test 55
3.3.4 Header Specific Test 58
3.3.5 Semantic Specific Test 61
3.4 Results 62
3.4.1 Experimental Setup 62
3.4.2 Fault Detection Performance 64
3.4.3 Overhead 68
3.5 Discussion 68
3.6 Conclusion 70

4 Verification of Power-based Side-channel Leakage through Simulation 71
4.1 Introduction 71
4.2 Power-based Side-channel Leakage 73

4.3 Side-channel Leakage Verification 75

4.4 Experimental Results 78

5 Pre-silicon Architecture Correlation Analysis (PACA): Identifying and Mitigating the Source of Side-channel Leakage at Gate-level 82

5.1 Introduction 82

5.2 Related Work 85

5.3 PACA Methodology for Identifying the Leaky Cells 88

5.3.1 Power Simulation 89

5.3.2 Selecting the Leakage Time Interval 90

5.3.3 Architecture Correlation for Computing Leakage Impact Factor . . . 92

5.4 PACA on encryption subcircuit 94

5.5 PACA on an AES hardware engine 96

5.6 PACA on PRESENT Hardware Engine 99

5.7 PACA of an SoC Bus Transfer 101

5.8 Selective Countermeasure with WDDL 104

5.8.1 Background in Circuit-level Countermeasures 105

5.8.2 Selective-replacement WDDL 106

5.8.3 Validation results 108

5.9 Selective Replacement Countermeasure with Decoupling Cell 109

5.9.1 Implementation of the Decoupling Cell 109

5.9.2 Selective Replacement Result 113
5.10 Discussion 115
5.11 Conclusion 120

6 Augmenting Leakage Detection with Boostrapping **121**
6.1 Introduction 121
6.2 Preliminaries 124
6.2.1 Leakage Detection using Welch's t-test. 124
6.2.2 The Bootstrapping Method. 126
6.2.3 Kolmogorov-Smirnov Test 127
6.3 Applying Bootstrapping to Leakage Detection 127
6.3.1 Simulating Leakage Detection 129
6.3.2 Experimental Results 131
6.4 Limitations 133
6.5 Implementation Details 135
6.6 Conclusion 138

7 Programmable RO (PRO): A Multipurpose Countermeasure against Side-channel and Fault Injection Attacks **140**
7.1 Introduction 140
7.2 Related Work 144

7.2.1 On-chip sensors as a countermeasure against power Side-Channel Analysis (SCA) . . . 145
7.2.2 On-chip sensors to detect/cause power perturbation . . . 145
7.2.3 On-chip sensors to detect fault injection . . . 146
7.2.4 Our contribution . . . 147
7.3 Programmable RO Design . . . 148
7.3.1 Background . . . 148
7.3.2 PRO Design and Configuration . . . 149
7.3.3 PRO Integration and Basic Principles . . . 152
7.4 Side-channel Countermeasure . . . 154
7.5 Power Sensing . . . 159
7.5.1 PRO Power Sensing with Regard to External Power Variations . . . 160
7.5.2 PRO Power Sensing with Regard to On-die Local Power Variations . 162
7.5.3 PRO Power Sensing with Regard to Sensor Locality . . . 164
7.6 Fault Detection . . . 165
7.6.1 Power Fault Detection . . . 166
7.6.2 Electromagnetic Fault Injection (EMFI) Detection . . . 170
7.7 Conclusion . . . 172
8 Overall Conclusion **175**

List of Figures

1.1 Basic Concept of Side-channel Analysis 4

1.2 Basic Setup of Side-channel Analysis 6

1.3 Basic Idea of CPA of attacking one key byte 7

1.4 CPA Results 10

1.5 Organization of Dissertation 13

2.1 Fault-assisted SCA methodology of masked software implementations. 19

2.2 Attacker Model adopted in this chapter. 20

2.3 (a) AES power trace and (b) its autocorrelation plot. 22

2.4 Byte-level Masked AES Implementation: (a) Pseudocode (b) Assembly code in RISC-V ISA. The red rectangles mark the fault injection targets. 24

2.5 Region of similarity: (a) Autocorrelation matrix of byte-level masked AES (b) Computed Region of Similarity (ROS). 27

2.6 Region of randomness: (a) Standard deviation for byte-level masked AES power traces with fixed plaintexts and (b) derived region of randomness. . . 29

2.7 Standard deviation $\sigma_2(t)$ after successful fault injection. 31

2.8 Experimental setup. 32

2.9 Clock glitching parameters. 32

2.10 Masked Bit-sliced Processor Allocation. 35

2.11 (a) Autocorrelation for bit-sliced masked AES power traces with fixed plaintext. (b) Derived regions of similarity. 37

2.12 (a) Standard deviation for bit-sliced masked AES power traces with fixed plaintext (b) Derived regions of randomness. 38

2.13 Standard deviation for bit-sliced masked AES power traces with fixed plaintext (a) before and (b) after disabling mask. 38

2.14 Welch's t-test for 16_*same*, 8_*same* and 4_*same*. 39

3.1 Fundamental Principles of Proposed Countermeasure 51

3.2 Three Levels in Methodology 53

3.3 Methodology 53

3.4 ABI specific fault detection wrapper structure 55

3.5 Header specific fault detection wrapper structure 60

3.6 Semantic specific fault detection wrapper structure 61

4.1 Visual comparison of power signals (grey) and correlation graphs (blue) for measurement, transistor-level simulation, gate-level simulation, and RTL simulation 78

4.2 Predictive quality of power simulation for side-channel leakage detection. This plot compares the leaky points of measurements and simulations at different levels of abstraction (X) and as a function of AES clock cycle (Y). 79

5.1 (a) Traditional side-channel leakage assessment flow. (b) Proposed PACA flow. 83

5.2 PACA flow for Identifying Leaky Cell 89

5.3 (a) AES sbox setup with Register Stages. (b) AES sbox setup without Registers. 95

5.4 Leakage peak for AES sbox with register stage setup. intermediate data = key_reg ⊕ text_in_reg 96

5.5 Leakage Time Interval for the AES hardware engine. Leakage Model: HD(AES state bit). 97

5.6 LIF Distribution for the AES hardware engine. Leakage Model: HD(AES state bit); Logarithmic Y scale. 98

5.7 Leakage Time Interval for the PRESENT hardware engine. Leakage Model: HD(PRESENT state bit). 99

5.8 LIF distribution for the PRESENT Hardware Engine. Leakage Model: HD(PRESENT state bit); Logarithmic Y scale. 100

5.9 SoC block diagram. 101

5.10 Leakage Time Interval for the SoC bus transfer. Leakage Model: HW(transferred bit). 102

5.11 LIF distribution for the SoC bus transfer. Leakage Model: HW(transferred bit); Logarithmic Y scale. 103

5.12 Selective-replacement WDDL (a) Original Circuit (b) Transformed Circuit (c) Clocking. 106

5.13 Impact on the Pearson Correlation Peak before and after replacing the two top-LIF cells by WDDL 107

5.14 Schematic of the decoupling unit. 110

5.15 Block diagram showing the placement of the decoupling cell. 111

5.16 Simulation results of the decoupling cell. 112

5.17 Impact on Pearson Correlation Peak before and after replacing only the Top-1 LIF cell by decoupling cell. 114

5.18 Correlation results for the AES Coprocessor using HD(AES state bit) obtained from (a) Simulated Traces, (b) ASIC Measurement Traces. 117

5.19 Correlation results for the SoC Bus Transfer using HW(transferred bit) obtained from (a) Simulated Traces, (b) ASIC Measurement Traces. 118

6.1 Bootstrap Leakage Detection Enhancement 128

6.2 The evolution of the p-value with increasing number of traces for TVLA (left) and with bootstrapping (right) using simulated traces 130

6.3 The sample distribution of the p-values taken from 5000 iterations of the bootstrapping method applied to samples where a the null hypothesis is false (left) and true (right) 131

6.4 The evolution of the p-value with increasing number of traces for TVLA (left) and with bootstrapping (right) applied to an implementation of AES in software 132

6.5 The evolution of the p-value with increasing number of traces for TVLA (left) and with bootstrapping (right) applied to an unpotected implementation of AES on an FPGA 132

6.6 The evolution of the p-value with increasing number of traces for TVLA (left) and with bootstrapping (right) 133

6.7 The negative log of p-value returned by the TVLA test for a fixed-versus-random t-test with 50000 traces (top left), 1000 traces with 20 iterations of the bootstrapping method (top right), 5000 traces with 5 iterations of the bootstrapping method (bottom left) and 50000 traces with 5 iterations of the bootstrapping method (bottom right) 135

7.1 PRO based on-chip Secure Network Hardware Extension 141

7.2 Propagation delay of a ring oscillator. 148

7.3 PRO Design. D0 donates the delay cell type-0, D1 donates the delay cell type-1, D2 donates the delay cell type-2. 152

7.4 Basic principles for PRO fault detection 153

7.5 Experimental Setup for Evaluating RO's performance in side-channel leakage hiding 155

7.6 AES power traces when PRO is (a) Off; (b) On; 156

7.7 Power Spectrum for power traces when (a) PRO off; (b) PRO on without driving IO pin; (c) PRO with fixed oscillation frequency and driving IO pin; (d) PRO with random oscillation frequency and driving IO pin; 157

7.8 T-value Comparison when PRO is on/off 158

7.9 Experimental Setup for PRO frequency changing as a function of external power supply 161

7.10 PRO's oscillation frequency with Regard of External Power Supply Voltage . 162

7.11 The structure of the employed RO-based power wasters. 163

7.12 Experimental Setup for PRO Power Sensing with Regard to On-die Local Power Variations 163

7.13 PRO Frequency with Regard of Local Power Supply 164

7.14 FPGA Floorplan for Evaluating PRO Performance with Regard of Sensor Locality 166

7.15 PRO's average Frequency Drop Ratio for each row versus the spatial proximity of the power wasters 167

7.16 Experimental Setup for PRO Power Fault Detection 168

7.17 FPGA floorplan for Evaluating PRO Performance in Power Fault Detection, power wasters simulate local power fault happens at location-1. 169

7.18 PRO average frequency drop ratio for each row when power fault happens at location-1 170

7.19 Floorplan and corresponding PRO average frequency drop ratio for each row when power fault happens at location-2. Black blocks donate PROs in the floorplan, red blocks donate power wasters positions in the floorplan. 170

7.20 Floorplan and the corresponding PRO average frequency drop ratio for each row when power fault happens at location-3. Black blocks denote PROs in the floorplan, red blocks denote power wasters positions in the floorplan. . . 171

7.21 Experimental Setup for PRO EM Fault Detection 173

7.22 Influence on the Frequency Distribution, X-axis is probability and Y-axis is frequency. 174

7.23 Influence on the Frequency Distribution for PRO-32 174

List of Tables

2.1 Real-world Pseudo Random Number Generation Functions 23

2.2 Number of Power Traces Needed to Retrieve Key 32

2.3 Implementation Parameters 35

2.4 Number of Traces Needed to Retrieve Key Bytes 39

2.5 Fault Injection Success Rate 40

3.1 Unlikely Value Example 54

3.2 Fault Detection Wrapper Response Cases 66

3.3 Test Results for RNG( ) Function Call 66

3.4 Test Results for memmove( ) Function Call 67

3.5 Overhead in Code Size 67

3.6 Overhead in Cycle Count 67

3.7 Estimated Overhead of Related Work 68

4.1 Abstraction Levels for Side-channel Leakage Verification 77

4.2 Simulation Performance in function of Abstraction Level 81

5.1 Pearson Correlation Threshold Levels as a Function Confidence 92

5.2 Example of Architecture Correlation 93

5.3 LIF Distribution Data for AES sbox with register stage setup 96

5.4 LIF Distribution Data for AES sbox without registers stage setup 97

5.5 LIF Distribution Data for the AES Hardware Engine using HD (AES state bit) as the leakage model 97

5.6 Runtime Evaluation for AES Hardware Engine (9,585 cells) 98

5.7 LIF Distribution Data for the PRESENT Hardware Engine using HD (PRESENT state bit) as the leakage model 100

5.8 Runtime Evaluation for PRESENT Hardware Engine (653 cells) 101

5.9 LIF Distribution Data for the SoC Bus Transfer Leakage Model: HW(transferred bit) 102

5.10 Runtime Evaluation for SoC Bus Transfer (99,904 cells) 104

5.11 Impact on the Pearson Correlation Peak under various levels of replacement 108

5.12 Power Simulation Levels Trade-offs 119

7.1 Configurations for PRO 151

Chapter 1

Introduction

An Embedded system is a computer system that has specialized functionality. It works either as an independent system or as a part of a large system. An embedded system includes a microprocessor or microcontroller, memory, and IO peripherals. The complexity of an embedded system ranges from a single microcontroller such as a smart card to a very large system that integrates multiple units, peripherals, and networks, such as hybrid cars. Embedded systems are an indispensable part of our daily lives, they are present in everyday things including but not limited to smart cards, smartphones, and automobiles which help achieve a convenient and modern life. Within embedded systems, private information is collected and stored, such as social security numbers, medical histories, addresses, and financial information. This information is used for personal communication, for financial transactions, to monitor patient health, to control home devices, etc and could be sabotaged by the malicious adversaries. It is has been repeatedly observed in the real world that significant cost will be paid if there exist insecurity in the embedded system and the security aspect has become a big concern for the user of embedded devices. For example, researchers have demonstrated that the transponder key of a car immobilizer can be recovered by a profiled side channel attack [151]. It is critical to guarantee the security of the embedded system.

There are several common security requirements to be considered when designing the embedded system [88]:

Secure Communication: While transferring the sensitive information, protecting the secret data from attack's snoop (Data Confidentiality), ensuring the data cannot be changed in transit by an adversary (Data Integrity), avoiding sensitive data from been sent/received by an unauthorized user (User Authentication);

User Identification: The only authorized user can have access to the embedded system;

Secure Network Access: Verify the identity of the device before granting its access to a network or a service.

Availability: Making sure the embedded system can perform expected functionality. Preventing adversaries from performing attacks that will harm the embedded system's performance, quality of service, etc.

Secure Storage: Protecting the secure storage(including external/internal storage devices) and content security of the sensitive data.

Tamper Resistance: Ensuring the system won't be maliciously probed by the adversary.

Cryptography, as security primitive is proposed to achieve security within the embedded system. It is a security mechanism which is based on mathematics and algorithms, among which computational hardness assumptions are of particular importance. Computational hardness assumes that adversaries only have limited computation resources and it is computationally infeasible to break the cryptographic primitives even though it is theoretically possible [116]. Computational security is the core idea of modern cryptography. There exists certain mechanisms, such as one-time pad [13], which can achieve information-theoretical security, i.e. the mechanism is still provably unbreakable even with unlimited computations resources. However, those mechanisms are difficult to use in practice due to their vast overhead.

Even though Crypto-systems are designed with the objectives to protect secret data they are still subject to attacks. Adversaries exploit the encryption/decryption procedure of cryp-

tosystem to learn the secret messages or secret keys. There are several attacks that retrieve the secret information in the crypto-systems with varying level of effectiveness. A brute force attack is an attack where the adversary exhaustively tries all possible secret key guesses and monitors the output of the crypto-system to check if any key guess results in a correct output. Brute-Force attack is not effective to modern crypto-systems mainly because of the computational hardness of crypto-algorithms. Computational hardness is typically achieved by increasing key length. For example, when an adversary is attacking an AES-128 cipher with brute force, there are 2^{128} key guesses which is impossible for existing general purpose computers to break AES-128 within a reasonable amount of time. Man-in-the-Middle (MIM) attack is another attack, which targets public key cryptosystems [104]. In MIM attack, the adversary inserts themselves into the communication channel by masquerading as a legitimate entity. The original parties believe that they are exchanging keys with each other, but instead, they end up exchanging keys with the adversary. Public-key crypto-systems like RSA (Rivest–Shamir–Adleman) can be used for authentication of the communicating parties before exchanging keys. There exists other traditional attacks such as replay attack, chosen plaintext attack, birthday attack, etc. The modern cryptographic algorithms have been developed and improved to be mathematically strong and resistant against these traditional attacks.

In recent years, implementation attacks, which target physical implementations of cryptographic algorithms, are gaining more and more attention. Implementation attacks are powerful since they can break a cryptographic algorithm even if it is computational secure. In implementation attacks, the adversaries have physical access to the embedded device which allows them to monitor or tamper with the activities of the embedded devices while running cryptographic algorithms. Side-channel attacks is one major implementation attack. It has become a serious threat to the security of embedded systems.

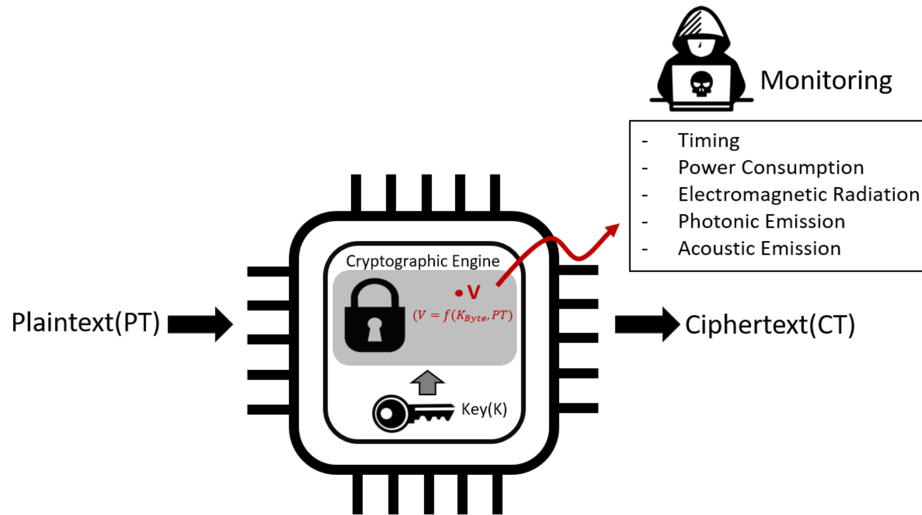

Figure 1.1: Basic Concept of Side-channel Analysis

1.1 Side-channel Analysis of Embedded Systems

In this section, we provide an introduction to the side-channel attack to the embedded system. As aforementioned, the security of embedded system's cryptographic operations is based on computational hardness, i.e. cryptographic algorithms are designed to be computationally infeasible to be broken. Take the 128-bit AES algorithm as an example, if an attacker wants to break the algorithm with brute-force, it means 2^{128} key possibilities need to be verified. Even if the attacker can check 1 billion AES keys per second with extremely powerful computing resources, almost 10^{13} years are needed to break the AES algorithm (i.e. 10,000 times longer than the age of the universe [9]). However, computational hardness is not able to guarantee the security of the crypto-algorithms anymore when side-channel analysis comes into scope. An attacker can break a crypto-algorithm much faster with side-channel analysis. Instead of looking at the whole key length as search space, the side-channel analysis uses the divide-and-conquer strategy and reveals the secret information piece by piece (byte-by-byte). Take 128-bit AES as an example. Side-channel analysis breaks one key byte

at a time and for each key byte, the search space is 2^8. Thus the search space for the whole 16 key bytes is $16 \times 2^8 = 2^{12}$. Compares to brute force, the computation complexity to break the key drops dramatically.

Figure 1.1 shows the basic concept of side-channel analysis. When a crypto-algorithm is running on the embedded systems, it takes in the plaintext (PT) as input, following the crypto-algorithm, it encrypts the plaintext with the secret key (K) stored on-chip and outputs the ciphertext (CT). As an attacker, the goal is to reveal the value of the secret key.

During the encryption/decryption operation, there exists certain intermediate value V whose value depends on the one byte of the secret key K and the input plaintext PT and follows:

$$V = f(K_{Byte}, PT) \tag{1.1}$$

This intermediate value V is not directly observable by the attacker. However, the value of V affects the external observable physical effects of the device. The physical effects typically include power consumption, electromagnetic (EM) emission, acoustics, optical radiation, timing, etc. These physical effects indicate the value of V and can be used by the attacker to infer the secret information. In a typical side-channel analysis, the attack monitoring the physical effect and taking the corresponding measurements (traces). By applying statistical analysis to the measurements, an adversary can derive the secret variables.

There are different side-channel analysis techniques, including Differential Power Analysis (DPA) [87], Correlation Power Analysis (CPA) [29], and Mutual Information Analysis (MIA) [65].

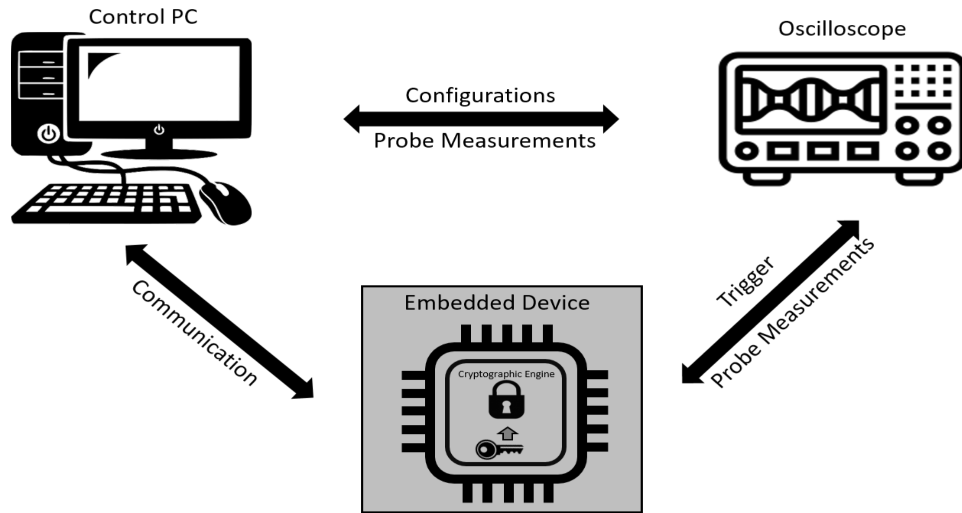

Figure 1.2: Basic Setup of Side-channel Analysis

1.1.1 Example of Side-channel Analysis

In order to explain the side-channel analysis procedure better, we take the 128-bit AES as an example and walk through the procedures of the Correlation Power Analysis (CPA).

Collecting Measurements

Figure 1.2 shows a common structure of side-channel analysis setup. The cryptographic algorithm is executed on the processor of the embedded device and the key is stored in the on-chip memory. The PC is used to communicate and control the target embedded device, e.g. starting/ending of the cryptographic program, sending in plaintext, reading out ciphertext, etc. There will be a probe to take the side-channel measurements of the embedded device. The probe can be a current probe for measuring the power consumption of the chip or an EM probe for measuring the EM radiation, etc. The embedded device also provides the trigger signal for the oscilloscope to control the starting of the trace collection. CPA requires a number of the measurements corresponding to different (random) plaintext

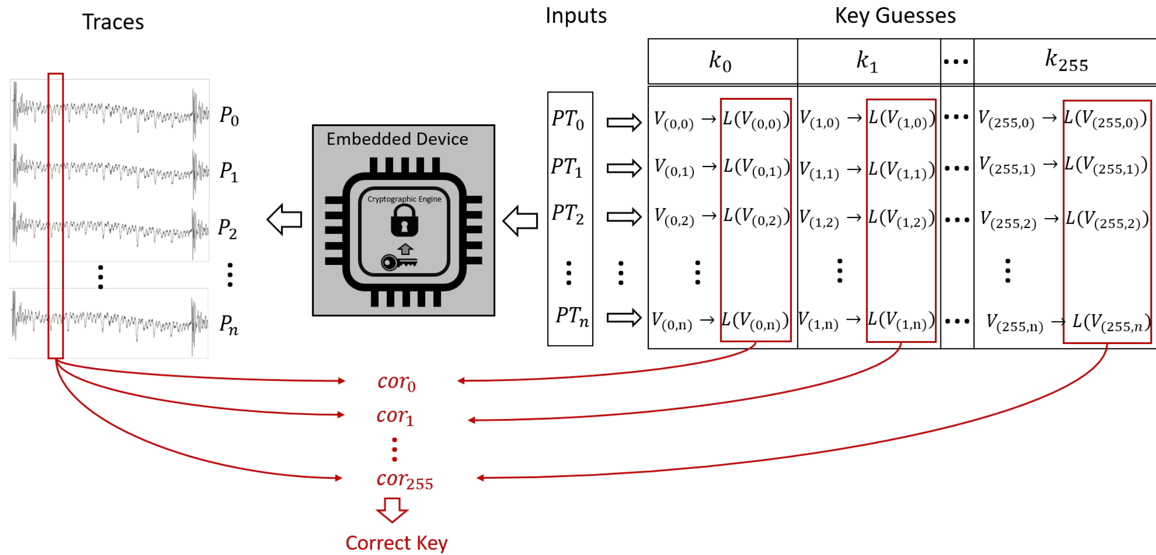

Figure 1.3: Basic Idea of CPA of attacking one key byte

inputs. Therefore, the PC needs to send the plaintext to the embedded device multiple times. After the plaintext is received by the device, the AES engine will start encryption and at the same time send the trigger signal to the oscilloscope to start the trace collection. After the trace collection is done, the measurements will be sent back to the PC for further analysis.

Analysis of Measurements

After obtaining all the measurements, the attacker can start analyzing the data and retrieve the key from it. The analysis requires a **Leakage Model**. The Leakage Model is a function computed over the secret intermediate value V (as defined in Equation 1.1). The objective of side-channel analysis is to reveal the value of V through many measurements and correlating those observations with $L(V)$. Popular choices for $L(V)$ are the Hamming Weight or the Hamming Distance on V; the Hamming Weight reflects value-based power leakage in CMOS, while the Hamming Distance reflects distance-based power leakage in CMOS.

Figure 1.3 demonstrate the basic idea of CPA for attacking one key byte. For each key byte, there will be 2^8 possibilities. The attacker first calculates a set of the intermediate value $V_{(i,j)}$ for ith plaintext input and jth key guess. The V is calculated based on Equation 1.1. Note that there are plenty of intermediate values that can be used by the attacker, for example, the intermediate value at the output of the first round's `Subbyte` operation is a popular choice by attackers. Next, for each intermediate value $V_{(i,j)}$, the attacker will calculate its corresponding leakage model value $L(V_{(i,j)})$. Suppose there are n plaintext inputs, we denote the Leakage Model value set for all the plaintext inputs with jth key guess as:

$$L(V_{(:,j)}) = \{L(V_{(0,j)})\ , L(V_{(1,j)})\ , L(V_{(2,j)}) \cdots L(V_{(n,j)})\} \tag{1.2}$$

Next, with the measurements and the leakage model, the attacker can use correlation analysis to distinguish the correct key guess from 256 possibilities (k_0 - k_{255}). We denote measured trace for ith plaintext input as P_i. The attacker first selects an attacking window cross all the measurements where the leakage may happen. Suppose the attacking window includes m sample points. At time point t on trace P_i, the corresponding measured value (sample point) is presented as $P_{(i,t)}$. We denote the measurement value set at time point t for all the plaintext input as:

$$P_{(:,t)} = \{P_{(0,t)}, P_{(1,t)}, P_{(2,t)} \cdots P_(n,t)\} \tag{1.3}$$

For key guess K_j and the time point t, we calculate the corresponding correlation coefficient as follows:

$$cor_{(j,t)} = \frac{cov(L(V_{(:,j)}), P_{(:,t)})}{\sigma_{L(V_{(:,j)})}\sigma_{P(:,t)}} \tag{1.4}$$

where:

cov = the covariance

$\sigma_{L(V_{(:,j)})}$ = the standard deviation of $L(V_{(:,j)})$

$\sigma_{P(:,t)}$ = the standard deviation of $P_{(:,t)}$

Iterating the same calculation through all the m sample points within the time window, a correlation coefficient trace corresponds to the key guess K_j will be generated. Then we grab the peak correlation coefficient cor_j value for the key guess k_j as follows:

$$cor_j = Max(cor_{(j,t)}) \tag{1.5}$$

where:

Max = the maximum value

We calculate the peak correlation coefficient (cor_0 - cor_{255}) for all the key guesses (k_0 - k_{255}) respectively. The maximum correlation coefficient value is supposed to point to the correct key guess. Figure 1.4 shows an example of the correlation coefficient traces. The correlation coefficient value of the correct key guess (highlighted in red) stands out from all the other wrong key guesses (in grey). This indicates that the attack on this key byte is successful. The attacker will repeat the same procedure for all the 16 key bytes to reveal the value of the secret key.

1.1.2 Existing Side-channel Countermeasures

Countermeasures against side-channel attacks eliminate or reduce the dependencies between the aforementioned physical effects and secret information. Masking and hiding are two popular techniques for side-channel countermeasures. In masking, each secret variable is split

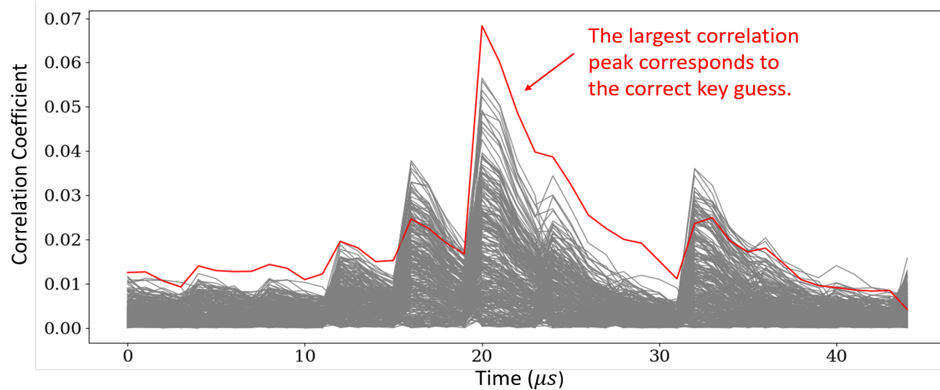

Figure 1.4: CPA Results

into two or more shares which are concealed by random numbers [38]. The side-channel leakage of each individual share does not reveal the secret variable because of the randomization introduced by random numbers. A random source which provides fresh random variables is significantly important in masking implementations. Threshold implementations extend the idea of masking while paying attention to glitches [114]. However, a generic architecture transformation technique that is low-cost and that deals with non-linear circuit effects remains elusive. Threshold implementation requires extra randomness which will cause other issues regarding how much randomness is needed, how frequent the random number needs to be refreshed, etc. Hiding countermeasures reduce the signal-to-noise ratio (SNR) for secret data-dependent operations. Hiding can be achieved by several techniques, such as shuffling which executes cryptography operations in random order [141], inserting random delays [36], and running multiple tasks in parallel [139].

1.2 Secure Embedded System Design Challenges

Side-channel attacks pose a practical threat to the security of embedded devices. As demonstrated previously, the embedded device's cryptosystem can be easily broken without ad-

equate protection against side-channel analysis. However, providing comprehensive side-channel protection to the embedded system is a very challenging task. An embedded system contains a large number of components. A modern embedded system integrates a hardware coprocessor to perform cryptographic operations, which is the primary source of side-channel leakage. However, even if the coprocessor implementation is protected with countermeasures, in the embedded system, the coprocessor is integrated together with other elements e.g. caches, memory hierarchy, interconnect infrastructure, and peripherals that participate in the cryptographic operation and hence can also leak side-channel information.

Even though the designer can make sure each individual component within the embedded system doesn't have side-channel leakage, the integration of each security component will possibly introduce new side-channel vulnerabilities. Therefore, it is essential to understand the effects of the integration within the embedded systems and investigating related attack models. However, this hasn't been well analyzed in the literature yet.

On the other hand, current research work in this field usually follows the pattern that the researchers first find vulnerabilities through investigating new attacks and then find corresponding solutions to the vulnerabilities. However, there are lots of vulnerabilities in the design known/unknown by the designers, how can we efficiently locate them and mitigate them? It is critical for the researchers to develop a novel side-channel leakage assessment mechanism to give designers metrics and guidance to address the root cause of the side-channel vulnerabilities within the design.

Furthermore, the side-channel countermeasures will generally introduce a large overhead, when it comes to a complex embedded system, it is almost infeasible to add countermeasure to the whole system. Therefore, precisely locating the side-channel leakage source is very important for the designers in order to fix the side-channel leakage. Additionally, locating the root cost of side-channel leakage in the design is also crucial for the designer to applied localized countermeasure with minimum overhead. However, this task is hard because of

the massive complexity of the embedded system. The task of examining the architectural elements of the embedded system for sources of side-channel leakage is precluded by the immense number of cells present in the embedded system. As the complexity and the number of IP blocks in an embedded system increases, the precise identification of the side channel leakage sources becomes arduous.

To make the challenge of building a secure embedded system even harder, in recent years, researchers have also further demonstrated that the Power Distribution Network (PDN) where the hardware design is located also brings in side-channel and fault attack vulnerabilities to the system. It has been demonstrated that attackers can place malicious circuits on a shared PDN with the victim chip. This enables the attacker to intrinsically inject malicious perturbations or monitoring changes of the victim circuit. Therefore, in order to guarantee the security of the PDN, a monitoring sensor network on the Power Distribution Network (PDN) should be built to detect such ongoing attacks. The monitoring network needs to fulfill the requirements including large spatial coverage, i.e., covering the full PDN area, and large temporal coverage, i.e., continuously monitoring the PDN [140].

1.3 Contributions of this Dissertation and Outline

In this dissertation, we demonstrate our contributions in addressing the challenges with regard of the integration of the security components and side-channel leakage root cause analysis in the procedure of embedded system design. Figure 1.5 shows the organization of this dissertation and we summarize our major contribution as follows:

Introduction	**Chapter-1**: Introduction
Investigating effect of Integration and Countermeasure	**Chapter-2**: Fault-assisted side-channel analysis of masked implementations
	Chapter-3: A Low-cost Function Call Protection Mechanism Against Instruction Skip Fault Attacks
Pre-silicon side-channel leakage evaluation and root cause analysis	**Chapter-4**: Verification of Power-based Side-channel Leakage through Simulation
	Chapter-5: Pre-silicon Architecture Correlation Analysis (PACA): Identifying and Mitigating the Source of Side-channel Leakage at Gate-level.
	Chapter-6: Augmenting Leakage Detection Using Bootstrapping
On-chip security monitoring extension	**Chapter-7**: Programmable RO (PRO): A Multipurpose Countermeasure against Side-channel and Fault Injection Attack
Conclusion	**Chapter-8**: Conclusion

Figure 1.5: Organization of Dissertation

1. Investigating the effects of integration of security components and developing corresponding countermeasures.

Masking is an important side-channel countermeasure technique that uses random masks to split sensitive cryptography variables into multiple shares. In masked implementation, secret value is concealed by random number to eliminate the dependency between the side-channel leakage and sensitive variables. In order to guarantee the resistance of the masking scheme against side-channel analysis, the direction of state-state-of-the-art research in embedded system security is focused on building powerful random number generators or strong masked ciphers. But it fails to notice that when integrating these secured components together, new vulnerabilities are introduced. In this chapter, we point out that the random number transfer procedure is an unprotected weak link which can be exploited by the attacker to easily hack the system. In the actual implementation, the random number transfer procedure is normally achieved by a function call. This link can be bypassed by a targeted glitch disturbance (fault injection). We demonstrate that integration of security components, which is largely ignored, is challenging and illustrate that techniques to the state-of-the-art masking

implementations are deceptively strong. This work will be presented in the Chapter-2.

To solve this issue, we propose a light-weight protection scheme to protect function calls from instruction skip fault attacks. Our proposed methodology is based on a basic feature of ABI(Application Binary Interface) which defines the calling conventions for all functions. ABI defines that a function's caller and callee always agree on a specific memory location or registers to which the return values are passed. We turn this specific location into a monitor to detect whether a function call was skipped or not. We demonstrate this protection scheme on a SPARC architecture and show that we can effectively protect function calls with low overhead when compared to existing countermeasures. This work will be presented in Chapter-3.

2. Pre-silicon side-channel leakage evaluation and root cause analysis

In current practice, the method to evaluate side-channel security vulnerabilities occurs after the chip tape-out. The designer always measures the prototype of the chip. However, once even a side-channel leakage is detected, it will be too late to fix the vulnerabilities. Therefore, building up simulation-based side-channel leakages assessment is crucial for enabling the designer to evaluate the side-channel vulnerability of the design at the pre-silicon stage. In this chapter, we first looked into the different circuit effects which cause the power-based side-channel leakage and we evaluate each simulation abstract level in terms of capturing the side-channel leakage. We will demonstrate this contribution in Chapter-4.

After investigating the side-channel leakage detection with simulation at early design stage, it is important to precisely locate the leakage source so that the designer can provide targeted countermeasures. In this chapter, we developed a gate-level netlist analysis methodology - Presilicon Architecture Correlation Analysis (PACA) that enables designers to precisely

identify the source of side-channel leakage in a design at the granularity of a single cell. PACA operates on the pre-silicon design description. Using experimental results from a practical SoC design and multiple case studies, we show that only a small number of cells are significantly contributing to side-channel leakage.

Based on PACA, we further propose **selective replacement** as a low-cost side-channel countermeasure. By protecting only the most leaky cells in a design, the overall side-channel leakage can be significantly reduced, and at a very low cost. We demonstrate selective replacement on an AES Sbox, where we replace the leaky cells identified by PACA with side-channel protected cells. We investigated the effects of two cell-level countermeasures including WDDL [143] and the countermeasures that use internal energy buffering [68]. This contribution will be presented in Chapter-5.

Alongside the aforementioned contributions in pre-silicon side-channel leakage assessment, we also investigate an improvement on existing side-channel leakage assessment- Text Vector Leakage Assessment (TVLA). In this research, we propose an extension of the current TVLA based on the bootstrap method which improves the efficiency of data usage, or looking at it from a different angle, significantly decrease the number of measurements needed for detecting the leakage. This Bootstrap work will be shown in Chapter-6

3. Developing on-chip side-channel and fault Monitoring Extension

In order to further guarantee the security of Power Distribution Network (PDN), we developed a multipurpose ring oscillator design - Programmable Ring Oscillator (PRO). As an integrated primitive, PRO can provide on-chip side-channel resistance, power monitoring, and fault detection capabilities to a secure design. We present a grid of PROs monitoring the on-chip power network to detect anomalies. Such power anomalies may be caused by exter-

nal factors such as electromagnetic fault injection and power glitches, as well as by internal factors such as hardware Trojans. By monitoring the frequency of the ring oscillators, we are able to detect the on-chip power anomaly in time as well as in location. Moreover, we show that the PROs can also inject a random noise pattern into a design's power consumption. By randomly switching the frequency of a ring oscillator, the resulting power-noise pattern significantly reduces the power-based side-channel leakage of a cipher. We discuss the design of PRO and present measurement results on a Xilinx Spartan-6 FPGA prototype, and we show that side-channel and fault vulnerabilities can be addressed at a low cost by introducing PRO to the design. We conclude that PRO can serve as an application-independent, multipurpose countermeasure. Result of this research will be covered in Chapter-7.

Chapter 2

Fault-assisted Side-channel Analysis of Masked Implementations

In this chapter, we will present a novel side-channel attack model against the embedded system based on the vulnerabilities caused by the integration of the security components. This work has been published in 2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST) [153].

2.1 Introduction

Side-channel leakage is a known vulnerability of secure hardware and software implementations, caused by the physical effects of computing with secret variables. Modern implementations of cryptography need side-channel countermeasures to prevent an adversary from analyzing that leakage and from uncovering the secret variables that caused it. Among the proposed countermeasures, *masking* or randomization of side-channel leakage is a popular technique as it offers a sound and implementation-independent basis of correctness [77]. In a masking side-channel countermeasure, each secret variable is split into (at least) two randomly chosen shares. The side-channel leakage of a separate share does not reveal the secret variable, and in theory this guarantees that a correctly masked computation is secure against side-channel analysis. However, masking is still vulnerable against attacks that measure and

combine the side-channel leakage from multiple shares. This has driven recent research into high-order side-channel countermeasures, which increase the number of shares above the number that can be measured by the adversary [34, 70].

A sensitive variable is split into k random shares using $k-1$ random numbers and a masking method. In this chapter, we use the Boolean-masking method that uses exclusive-or to combine the sensitive variable v with a random number r_1, ..., r_{k-1} into k shares $[v \oplus r_1 \oplus ... \oplus r_{k-1}, r_1, ..., r_{k-1}]$.

The masking of an entire cryptographic algorithm requires a large amount of random numbers, which have to be generated locally and which have to be kept secret. These masks have to be continuously refreshed, since mask reuse is a known vulnerability. Non-linear operations (e.g. AND or modular multiplications) require additional mask signals or mask-dependent precomputed lookup tables. Finally, the amount of random numbers increases with the masking order. As a result, masked implementations need to use dedicated pseudo-random number generation algorithms, such as stream ciphers and block ciphers, to create a sufficient amount of randomness to mask the sensitive computations.

In this chapter, we demonstrate that the random-number-generation part is a weak link in the protection of masked algorithms. Using a well-placed fault injection, we can disable the masks from a masked algorithm. Once the mask is disabled, the masked algorithm becomes susceptible to a first-order side-channel attack.

2.1 summarizes the proposed attack methodology. First, we measure a set of power-based side-channel traces T from a masked algorithm. We analyze those traces to reveal the regions-of-similarity (ROS), the parts of the trace that most likely belong to the same algorithmic kernel. This helps to distinguish macro-level algorithm steps such as mask generation, key-schedule generation and encryption. Next, we also find the regions-of-randomness (ROR),

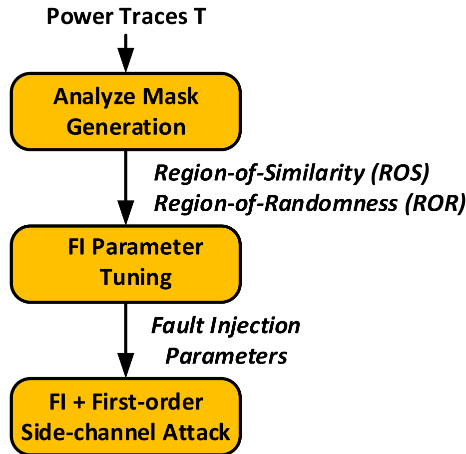

Figure 2.1: Fault-assisted SCA methodology of masked software implementations.

the parts of the trace that most likely correspond to mask generation. We use the ROS and ROR to tune the proper fault injection parameters. The objective is to identify the proper combination of fault intensity and fault injection time such that the mask is disabled, while the masked algorithm still computes. We demonstrate an adaptive algorithm that can test when the fault injection is successful, and we show that the estimation of ROR and ROS greatly reduce the search space of the fault injection parameters. Finally, with the proper fault injection parameters, the masked algorithm becomes vulnerable to first-order side-channel analysis.

The proposed methodology is independent of the masked algorithm. We apply the attack on two different masked AES implementations mapped on a RISC-V microprocessor. The first implementation is a first-order byte-level masked AES design. The second implementation is a first-order bit-sliced masked AES design. In each case, we demonstrate the methodology and achieve a successful side-channel attack.

2.2 Background

In this section we give a brief background on the underlying concepts of the proposed attack.

2.2.1 Attacker Model

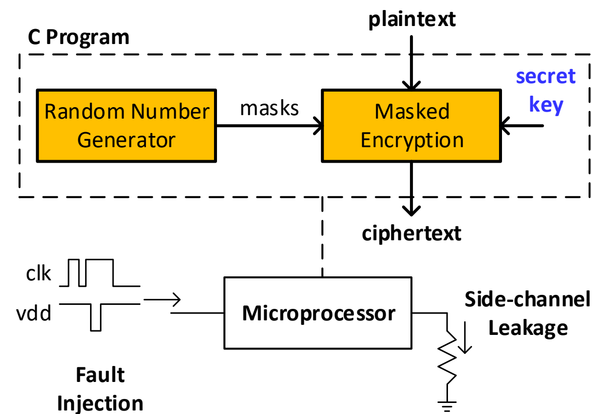

Figure 2.2: Attacker Model adopted in this chapter.

Figure 2.2 shows the attacker model. The target is a software masked algorithm, including encryption and a pseudo-random number generator. The software executes on a microprocessor. The adversary is able to monitor power-based side-channel leakage of the microprocessor, and the adversary can apply fault injection through clock glitching or power glitching. The adversary can control the input to the target. The adversary has no access to the internals of the microprocessor including its program-memory and its data-memory.

2.2.2 Signal Processing Tools

We use standard deviation and autocorrelation to post-process side-channel leakage. For completeness we define each of these metrics.

Standard deviation

For a given set of K power traces, we define the standard deviation over each time point t across the set of power traces.

$$s(t) = \sqrt{\frac{\sum_{i=1}^{K} (p(i,t) - \bar{p}(t))^2}{K-1}}$$

where $p(i,t)$ is the sample value of trace i at time t and $\bar{p}(t)$ is the average trace value at time t. The standard deviation is large when there is a large variation among the traces at time t. The analysis of standard deviation will help us to identify the presence of randomization.

Autocorrelation

Autocorrelation is the correlation between a signal and a time-shifted value of itself. Correlation peaks show up when there is a maximum overlap between the two signals. Therefore, autocorrelation can be used to identify repetitive patterns in a signal.

A convenient method to present the results of autocorrelation is a two-dimensional plot. A single power trace $p(t)$ is partitioned into L smaller fragments of $\frac{N}{L}$ samples each. Each of these fragments is then correlated with every other fragment, constructing the correlation matrix M such that M_{ij} contains the correlation between fragment i and fragment j. Finally, the matrix M is plotted as a gray-scale diagram, where a black pixel represents good correlation and a white pixel represents bad correlation.

In cryptographic implementations, repetitive computations (and repetitive side-channel leakage) are very common. For example, AES has ten encryption rounds and in each round, except for the last, the same sequence of `SubBytes`, `ShiftRows`, `MixColumn`, and `AddRoundKey` are executed. Each of these operations has internal repetition as well, and repeats a similar

operation for each byte of the state. The autocorrelation graph of AES can reveal these structures. Figure 2.3a presents the power trace of 128-bit AES encryption and its autocorrelation plot is shown in Figure 2.3b. We can recognize the key expansion, followed by 9 regular rounds and a final round with missing `MixColumn`. The autocorrelation matrix reveals repetitive patterns as correlation peaks away from the main diagonal. For example, each of the 9 regular rounds is similar, and the autocorrelation plot reveals this as a 9-by-9 block structure. It is clear that, even without insight into the details of the signal structure, autocorrelation is a powerful tool to reveal the macro-level structure of an algorithm.

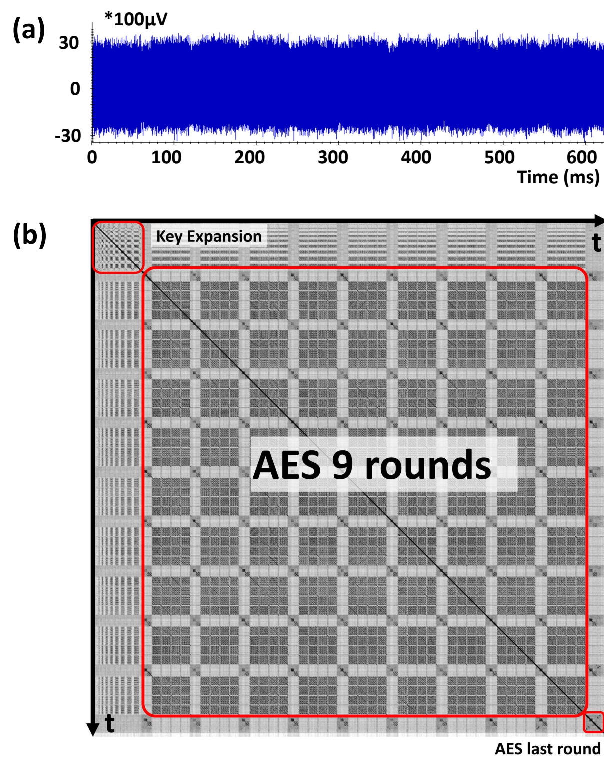

Figure 2.3: (a) AES power trace and (b) its autocorrelation plot.

Table 2.1: Real-world Pseudo Random Number Generation Functions

Library	Random Number Generation API	Output Transfer
OpenSSL [6]	`RAND_bytes(outBuf, nBytes)`	`memcpy(outBuf, intBuf, nBytes)`
mbedTLS [5]	`mbedtls_ctr_drbg_random(outBuf, nBytes)`	`memcpy(outBuf, intBuf, nBytes)`
Botan [3]	`randomize(outBuf, nBytes)`	`copy_mem(outBuf, intBuf, nBytes)`
Libgcrypt [4]	`gcry_randomize (outBuf, nBytes)`	`for (i=0; i<nBytes; i++) outBuf[i] = intBuf[i]`

2.3 Implementation of Masking Countermeasure

We now turn our attention to the practical implementation of a masking scheme. We first explain random number generation and a byte-level masked AES implementation. Next, we clarify the principal steps of the proposed attack.

The security of the masking countermeasure relies on frequently refreshed masks. As embedded systems do not have a reliable source of true random bits, the common practice is to employ Pseudo Random Number Generators (PRNGs) to generate masks.

A PRNG uses a deterministic algorithm to generate a random number stream from a seed of true entropy [81]. The PRNG output is unpredictable as long as the internal state of the algorithm is kept secret. For every invocation, the PRNG generates the requested amount of random numbers based on its internal state, and transfers the generated random numbers to an output buffer [41]. The PRNG also updates its internal state. PRNGs use encryption algorithms and hash functions in a repetitive fashion, and a PRNG's execution can hence be distinguished using autocorrelation of a power trace.

2.3.1 Pseudo Random Number Generation

Table 2.1 lists a set of example pseudo random number generators from several open-source cryptographic libraries. The first column of the table shows the application programming interface (API). Each PRNG in Table 2.1 takes the address (`outBuf`) and size (`nBytes`)

of an output buffer from the consuming application, generates the random numbers into an internal buffer `intBuf`, and finally, transfers the values kept in `intBuf` to `outBuf` by executing the transfer block. The `intBuf` isolates the internal state of the PRNG from the PRNG user, and prevents consuming applications from directly accessing the internal state of the PRNG. The second column of Table 2.1 shows how the PRNG transfers random numbers from the internal buffer to the output buffer.

In this chapter, we use a T-box implementation of AES-128 in counter mode to generate random numbers like the implementations in Table 2.1. The PRNG secret seed is the initial counter value, and the PRNG output is the ciphertext. The transfer operation is the function `memmove()`.

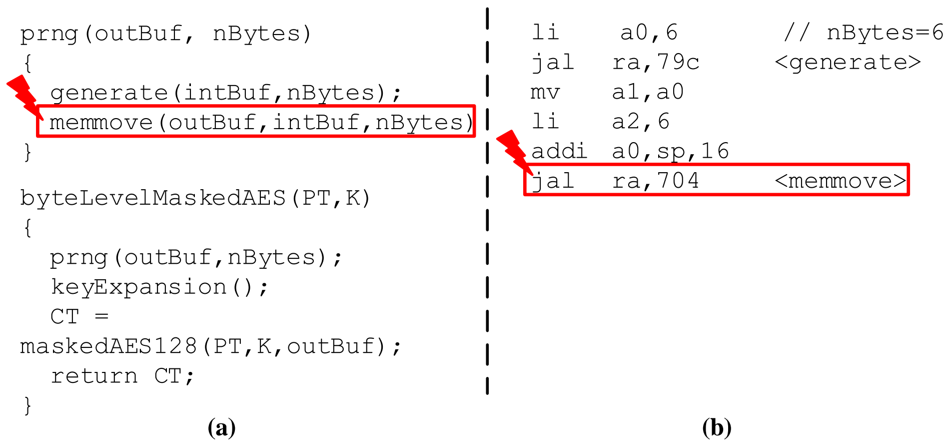

Figure 2.4: Byte-level Masked AES Implementation: (a) Pseudocode (b) Assembly code in RISC-V ISA. The red rectangles mark the fault injection targets.

2.3.2 Byte-level Masked AES Implementation

Figure 2.4a shows the pseudocode of our byte-level masked AES implementation. The `byteLevelMaskedAES()` function takes a 128-bit plaintext `PT` and key `K` as input. Then it produces the masks and round keys for the AES by calling `prng()` and `keyExpansion()`

functions, respectively. Our PRNG (`prng()`) first generates the required amount (`nBytes`) of random numbers in an internal buffer `intBuf`, and then transfers them to an output buffer `outBuf`. Finally, `byteLevelMaskedAES()` calls the masked AES function `maskedAES128()` to compute a 128-bit ciphertext `CT`. The `maskedAES128()` function applies 10 rounds of AES on the inputs `PT`, `K`, and masks stored in `outBuf`. We use a `maskedAES128()` based on the work of Mangard et al. [98].

2.3.3 Disabling the Mask

The proposed attack aims at disabling the transfer operation of fresh random masks from the PRNG to the consuming function. In software, this transfer is realized by means of a memory-move operation between two buffers (Table 2.1). The transfer operation is characterized by the amount of bytes `nBytes` being copied, the starting time `S` of the copy operation, and the time period `P` it takes to complete that operation. Hence, we define the **transfer operation** as `F(nBytes,S,P)`. The objective of the attack is to disable the successful completion of `F` by injection faults into the processor over a period `[S, S+P]`. While `S` and `P` are not known to the attacker, they can be estimated through the proposed attack method.

The red marking in Figure 2.4a symbolizes a fault attack on the transfer operation. A fault injection can cause an instruction of a microprocessor to be skipped, or to be ineffective. Figure 2.4b is a partial assembly listing of `byteLevelMasked`. A fault injected during the jump-and-link (`jal`) instruction skips the `memmove`, and disables the transfer operation. In this chapter, we use this fault model.

Algorithm 1: Find region of similarity (ROS)

Input : Autocorrelation matrix $M(1..n, 1..n)$
Output: Regions of similarity ROS

```
1  for i = 1 to n do
2  |   DiagSum(i) = sum_{j=1}^{j=i} M(j, i - j + 1)
3  end
4  Pks = Arg(LocalMax(DiagSum))
5  LeftEdge = 0
6  for j = 1 to Pks.length do
7  |   Width = 2 · (Pks_j - LeftEdge)
8  |   ROS_j.left = LeftEdge
9  |   ROS_j.right = ROS_j.left + Width
10 |   ROS_j.height = DiagSum(Pks(j))
11 |   LeftEdge = ROS_j.right
12 end
```

2.4 Methodology

This section explains the proposed methodology to attack masked algorithms. We illustrate the attack on a byte-level masked AES implementation.

2.4.1 Analyze Mask Generation

The first step is to locate the random-number transfer operation. Starting from a set of power traces, we identify portions of a power trace that are highly likely to correspond to random-number transfer. This is done through a two-step process. First, we partition the power trace into Regions of Similarity, which are portions that are highly likely to belong to the same algorithmic kernel. Next, we mark some Regions of Similarity as Regions of Randomness. These are the portions of the power trace that show a high variation from trace to trace, even when the algorithm input is kept constant.

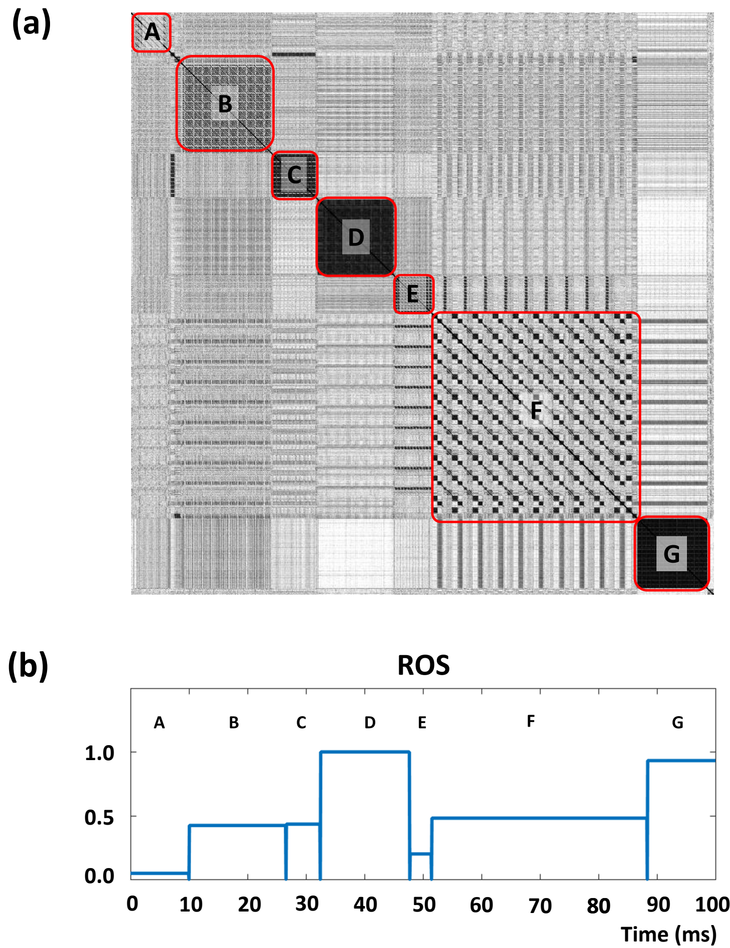

Figure 2.5: Region of similarity: (a) Autocorrelation matrix of byte-level masked AES (b) Computed Region of Similarity (ROS).

Identifying the Regions of Similarity (ROS)

The purpose of this step is to reveal the overall structure of the masked algorithm, making use of the property that cryptographic algorithms use repetitive computations.

The Regions of Similarity (ROS) can be found from a single power trace, or from the average of multiple noisy traces, by computing and analyzing the autocorrelation matrix. Figure 2.5a shows the autocorrelation matrix of a byte-level masked AES algorithm. We can visually recognize distinct portions A through G in the plot. For automatic analysis of autocorrelation

matrix, we can use Algorithm 1 which produces the plot shown in Figure 2.5b. This algorithm first computes $DiagSum(i)$, the sum of the elements on the anti-diagonal of a submatrix $M(i, i)$. $DiagSum(i)$ reaches a maximum when the antidiagonal cuts through the center of a region of similarity in M. Therefore, a local peak detector on $DiagSum$ gives us the locations of the center of each region of similarity ROS_j. By growing each ROS_j symmetrically around the center, we find the ROS curve shown in Figure 2.5b. The height of each ROS_j is the local peak value of $DiagSum$.

Identifying the Regions of Randomness (ROR)

The second step is to identify which Region of Similarity is responsible for random number generation. To do this, we measure a set of power traces under constant input, such as under constant plain-text. Next, we compute the standard deviation over the resulting set of curves. Portions of the curve with a high standard deviation indicate random number generation activity, or masked computations that use internally generated random numbers.

Figure 2.6a shows the standard deviation trace $\sigma_1(t)$ for 500 power traces under fixed plain-text. We have marked A through G on the curve, and note that some portions have a high standard deviation while others have not. Algorithm 2 can then be used to mark what portions of the standard deviation should be considered random activity. We use the empirical rule that any standard deviation value that falls outside of two times the expected deviation (over the deviation) should be considered random. This leads to Figure 2.6b.

Candidates for Mask Transfer Operation

The combination of ROS and ROR leads to the set of candidate regions for random number generation. We mark regions A, D and F. Among those, we can discount F because the internal symmetry of F in the autocorrelation matrix indicates it is likely to be 10 rounds

Algorithm 2: Find region of randomness (ROR)

Input : Standard deviation vector $\sigma(t)$

Output: Regions of randomness ROR

```
1 σ̂ = standard_deviation(σ(t))
2 μ̂ = average(σ(t))
3 for i ∈ {1, ..., σ.length} do
4   | if σ(i) > μ̂ + 2 · σ̂ then
5   |   | ROR(i) = 1                          ▷ mark as random
6   | end
7 end
```

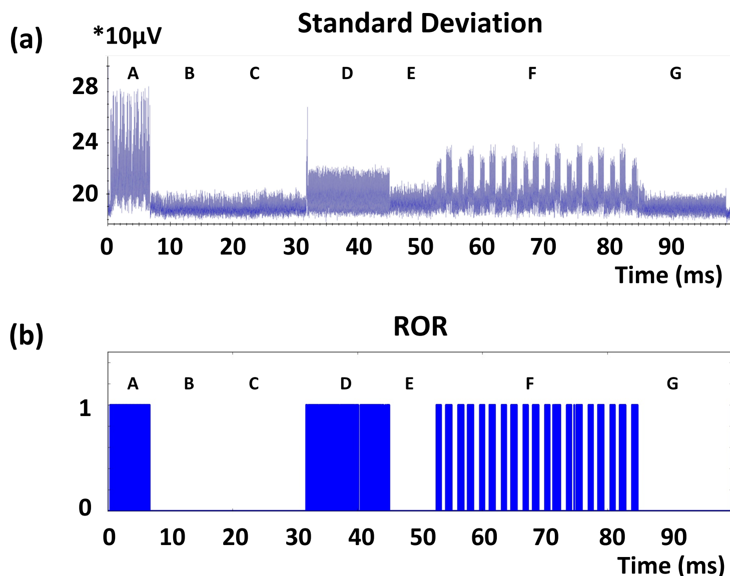

Figure 2.6: Region of randomness: (a) Standard deviation for byte-level masked AES power traces with fixed plaintexts and (b) derived region of randomness.

of AES. Thus, the only two remaining candidates would be A and D. The fault injection attack to disable the mask would aim at the transfer operation, which is located at the end of region A or D. Hence, through preprocessing and analysis, we can reduce the number of fault injection candidate locations enormously.

Algorithm 3: Tuning Glitch Position and Glitch Cycles

Input : Clock glitch parameter sets GW, GP and GC
Output: Successful clock glitch parameters gw, gp, gc

```
1  gw = GW_fs                                             ▷ fs: fault sensitivity
2  forall GP_i in GP and GC_j in GC do
3      T_fault-injection = ∅                              ▷ power trace set
4      for k = 1 to N do
5          Perform fault injection at GP_i with GC_j and gw
6          Collect the corresponding power trace t
7          if correct ciphertext then
8              Add power trace t to T_fault-injection
9          end
10     end
11     σ_2(t) = standard_deviation(T_fault-injection)
12     Φ = Σ_i (σ_1(i) − σ_2(i))
13     if Φ > Φ_max then
14         Φ_max = Φ, gp = GP_i, gc = GC_j
15     end
16 end
```

2.4.2 Tuning Fault Injection Parameters

The next step is to fine-tune the fault injection parameters to the precise settings that will eliminate the transfer operation of the random number generator. For the clock-glitch fault injection method introduced earlier, the fault injection parameters include the glitch position (GP), the glitch width (GW) and the number of glitch cycles (GC).

Initially the glitch width (GW) is adjusted to the fault sensitivity of the target system [93]. This is the *lowest* fault intensity needed for the system to be affected by fault injection. The glitch width can be set by repeatedly running a test algorithm (such as sum of numbers) while slowly decreasing the glitch width until we notice the program computes a faulty result.

We then need to find the other fault injection parameters GP and GC. We use side-channel leakage measurement, and the computation of standard deviation over the side-channel traces, to evaluate the success of the fault injection campaign. A successful fault injec-

tion would dramatically reduce the standard deviation without affecting the ciphertext. As demonstrated in Algorithm 3, faults are injected over the potential positions of random number transfer in the masked algorithm. Hence, we use the final portion of region A and D as fault target regions. We repeat each fault injection multiple (N) times for the same fault injection parameters, to obtain a set of fault power traces, for which the standard deviation σ_2 can be computed. Eventually, Algorithm 3 will select the fault injection parameters that cause the biggest reduction in standard deviation. 2.7 shows the dramatic standard deviation reduction after successful fault injection.

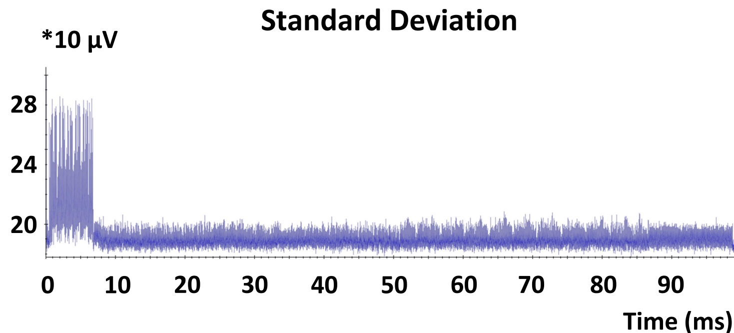

Figure 2.7: Standard deviation $\sigma_2(t)$ after successful fault injection.

2.4.3 Fault Injection and Differential Power Analysis

Using the fault injection parameters, fault injection is applied to skip mask transfer operation, thus disabling the mask. The power traces measured after fault injection can now by subject to first-order differential power analysis (DPA). In our experiment, we attack the Sbox output with a Hamming weight leakage model. Figure 2.2 shows that all key bytes are found from less than 300 traces.

Table 2.2: Number of Power Traces Needed to Retrieve Key

Number of key bytes retrieved	Number of traces
4	50
8	80
12	130
16	300

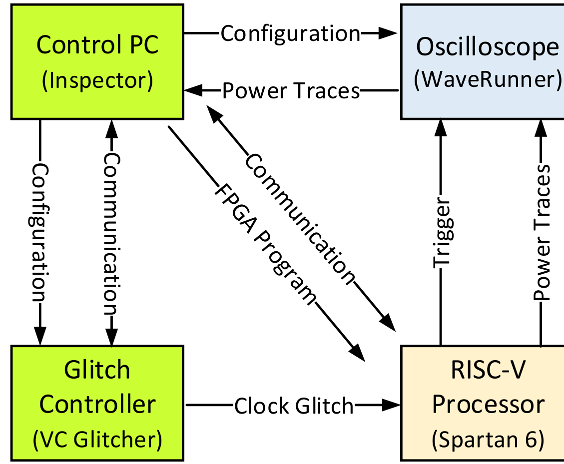

Figure 2.8: Experimental setup.

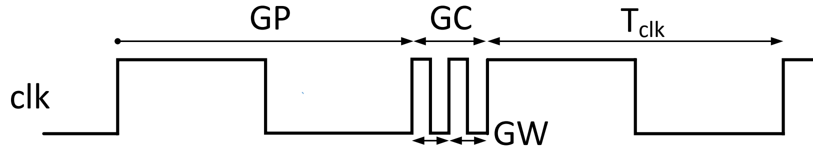

Figure 2.9: Clock glitching parameters.

2.5 Experimental Setup

The experimental setup in fig:setup shows a control PC, a glitch controller, an oscilloscope and a RISC-V processor. Masked AES software runs on the 32-bit RISC-V processor [8], configured in a Spartan-6 FPGA on a SAKURA-G board. The oscilloscope collects the power traces of RISC-V processor during AES execution. The Riscure VC Glitcher [7] performs fault injection through clock gliches. The control PC drives the overall configuration and

handles the post-processing of the collected power traces.

Figure 2.9 shows the effect of a glitch on the clock signal. A clock glitch temporarily shortens the clock period from T_{clk} to GW, thereby causing a setup time violation. We control the clock glitch injection with three parameters: the glitch position (GP), the glitch width (GW) and the number of glitches (GC). Our setup uses a minimum glitch width of 4 ns and a minimum step size of 2 ns.

2.6 Case Study

In the following case study we apply our methodology to attack a bit-sliced masked algorithm.

2.6.1 Bit-Sliced AES Implementation

Bit-sliced software design is a programming technique that treats an n-bit processor as n single-bit processor slices. The origins of bit-sliced software design are in high-throughput implementations of cryptographic algorithms [22, 79]. To develop a bit-sliced version of a cipher, a cipher is first expanded into sequence of single-bit operations. By expressing this sequence of bit-operations using bit-wise instructions, an n-bit processor will then execute n instances of the bit-sliced cipher in parallel. The bit-sliced algorithm thus operates on n blocks of data in parallel, and each block is processed bit by bit. Per-block input data is converted into bit-sliced format using a transpose operation. For example, a bit-sliced version of AES on a 32-bit processor will first transpose 32 input blocks of 128 bits each into 128 words, compute the AES bit-sliced algorithm, and finally inverse-transpose the 128 output words back into 32 output blocks.

2.6.2 Bit-Sliced masked AES Implementation

We developed a bit-sliced masked AES, which allocates the shares of a masked bit in slices within the same processor word. Thus, a 32-bit processor word contains 16 first-order masked slice pairs. Figure 2.10 shows how this masking is integrated as part of the transpose operation: 16 blocks of AES are converted into 128 words with masked bit-sliced data. As a countermeasure, this representation has the advantage that it prevents side-channel leakage from individual secret shares.

The masked version of the bit-sliced algorithm is developed as follows. First, observe that we only need to convert bitwise operations because it is a bit-sliced algorithm. For linear operations (such as `xor`, `not` and `mov`), the masked bit-sliced operations are identical to the unmasked bit-sliced operations. The non-linear `AND` operation is implemented using the secure-AND design of Ishai *et al.* [77]. Algorithm 4 demonstrates a secure-AND computed with bit-sliced masked data P and Q, and a fresh random mask R. The operations $\odot$ and $\oplus$ represent bitwise-and and bit-wise-xor respectively. The operation $rot()$ is a slice-rotation operator that swaps the even bits and the odd bits of a processor word. This is achieved using an expression such as `((v<<1) & 0xAAAAAAAA) | ((v>>1) & 0x55555555)`.

Algorithm 4: Order-1 masked bit-sliced AND

Require: $P = [p_1, p_0], Q = [q_1, q_0], R = [r, r]$
Ensure: $Y = sand(P, Q)$
1: $N_1 \leftarrow P \odot Q$
2: $N_2 \leftarrow rot(Q)$
3: $N_3 \leftarrow P \odot N_2$
4: $N_4 \leftarrow R \oplus N_1$
5: $Y \leftarrow N_4 \oplus N_3$
6: **return** Y

Table 2.3 compares the bit-sliced masked AES design with the byte-level masked AES design. The bit-sliced implementation consumes higher amounts of randomness because each bit

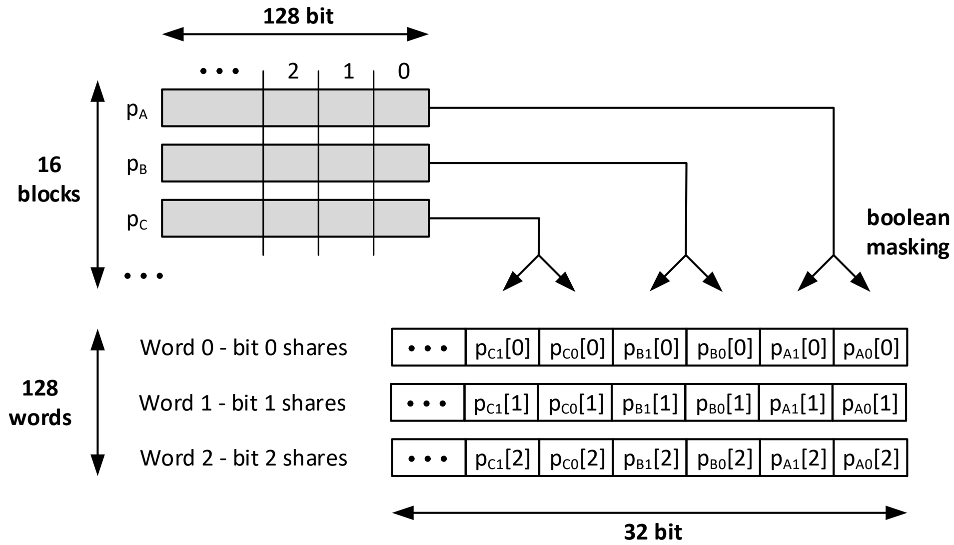

Figure 2.10: Masked Bit-sliced Processor Allocation.

Table 2.3: Implementation Parameters

Masked AES	**Byte-level**	**Bit-sliced**	Unit
Randomness	6	1,024	bytes
Program cycle count	86,080	7.328×10^6	cycles
Program size	12,172	15,968	bytes

of an AES block is masked with a separate random bit, and because the S-box step is implemented using logic expressions rather than using memory lookup tables. We found that open-source reference implementations of masked algorithms are not easy to find. To make our experiments verifiable, we have released the source code of our designs online [1].

2.6.3 Steps of methodology

Next, we explain how the methodology is used to attack bit- sliced masked AES.

[1]https://github.com/Secure-Embedded-Systems/Masked-AES-Implementation.

Analyze Mask Generation

First, we collect 500 power traces with fixed plaintext. Then we apply autocorrelation and standard deviation to locate the random number generation and transfer operation. Following the methodology, we obtain the autocorrelation matrix shown in Figure 2.11a. The ROS is found using Algorithm 1. The annotated result of Figure 2.11b identifies regions A through F. We then compute sample-wise standard deviation on the collected traces (Figure 2.12a). Using Algorithm 2, the resulting ROR is shown in Figure 2.12b. Finally, by checking the intersection of ROS and ROR, the region that is related to random number generation is narrowed down to A, C, D, E, F. However, region B has 10 repetitive patterns in the autocorrelation matrix. Region E is highly likely to be 10 rounds of AES. Also, since random numbers are generated before AES operation, we eliminated the region F. Therefore, the remaining candidates for random number generation are regions A, C, and D.

Tuning Fault Injection Parameters

In this step, we apply Algorithm 3 in regions A, C and D to tune the clock glitch parameters to skip the transfer operation without affecting the AES functionality. We successfully skip the transfer operation when we use the glitch width of 4 ns, glitch position of $570,600th$ cycle, and a single glitch cycle. This the glitch position is located near the end of region A. Figure 2.13 shows the dramatic standard deviation drop in the first two rounds of AES after fault injection. The standard deviation in region A is still high since it corresponds to random number generation.

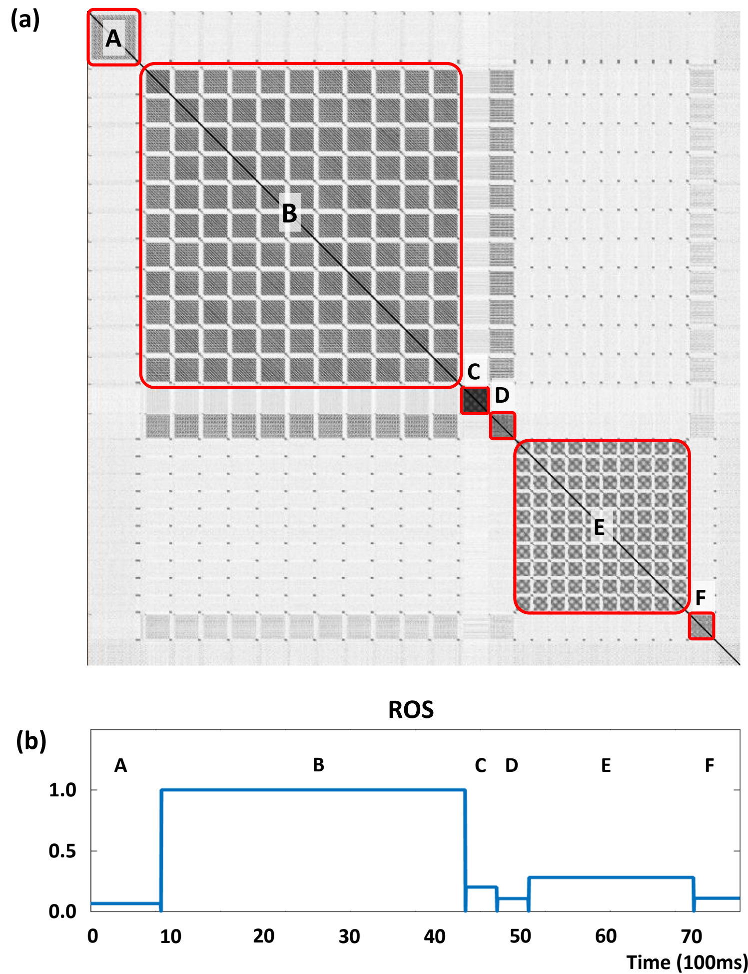

Figure 2.11: (a) Autocorrelation for bit-sliced masked AES power traces with fixed plaintext. (b) Derived regions of similarity.

Fault Injection and Differential Power Analysis

With the derived clock glitch parameters, clock glitches will be inserted to perform instruction skip on transfer operation. We collect power traces with random plaintext while clock glitch is inserted to the execution of masked implementation. Then, first-order DPA is used to attack the masked bit-sliced AES. We attack the Sbox output using the single-bit leakage model. The secret key can be retrieved byte by byte.

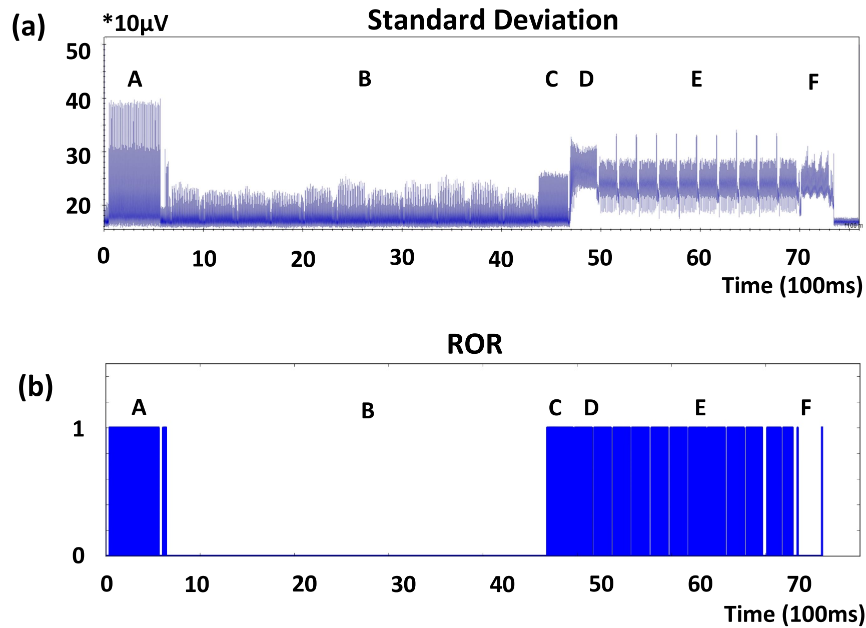

Figure 2.12: (a) Standard deviation for bit-sliced masked AES power traces with fixed plaintext (b) Derived regions of randomness.

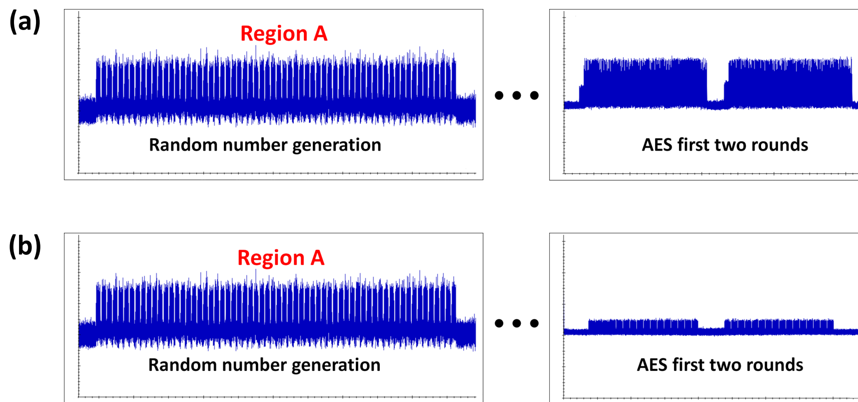

Figure 2.13: Standard deviation for bit-sliced masked AES power traces with fixed plaintext (a) before and (b) after disabling mask.

2.6.4 Results

We demonstrated our method on a bit-sliced AES implementation, which runs 16 instances of AES in parallel. We analyzed the efficiency of our DPA attack on three input configurations

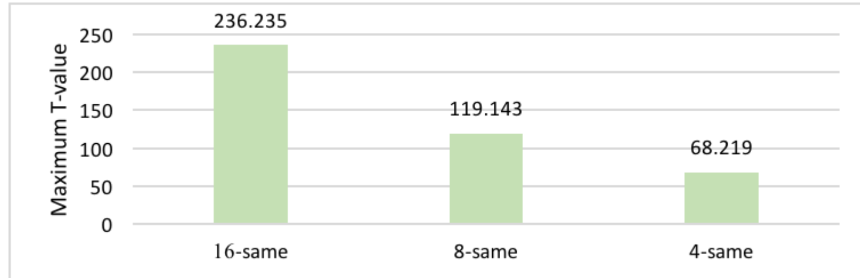

Figure 2.14: Welch's t-test for 16_*same*, 8_*same* and 4_*same*.

of the bit-sliced AES, where 16, 8 and 4 plaintexts are kept the same while the rest of plaintexts are random. We name them as 16_*same*, 8_*same* and 4_*same*, respectively. Table 2.4 lists the number of traces needed to retrieve different number of correct key bytes. As it is seen, the number of required traces increases as the number of random bit-slices increases. The reason behind this behavior is that more random bit-slices lead to more algorithmic noise. In addition, we provide Welch's t-test results for the three configurations of the bit-sliced AES in Figure2.14. The figure shows the maximum absolute T-statistic values calculated over 1000 power traces. Similar to the results in Table 2.4, 16_*same* has the highest leakage (i.e, the highest T-statistic value) due to lowest algorithmic noise. In addition, all T-statistic values shown in Figure 2.14 are larger than 4.5. This shows that all configurations leaks data through power side-channel.

Table 2.4: Number of Traces Needed to Retrieve Key Bytes

Number of key bytes retrieved	**Number of traces**		
	16_*same*	8_*same*	4_*same*
4	50	400	700
8	50	600	1100
12	100	900	1800
16	230	1800	4300

Table 2.5: Fault Injection Success Rate

Masked AES	**Number of Trials**	**Best Fault Injection Success Rate**
Byte-level Masked AES	6	100%
Bit-sliced Masked AES	5	100%

2.7 Discussion

In this section, we elaborate on fault injection success rate, related work and possible countermeasures against this attack.

2.7.1 Fault Injection Success Rate

Table 2.5 shows the number of fault injection trials and the best fault injection success rate obtained. We achieve 100% fault injection success rate. When countermeasures such as clock jittering would be present, the success rate may be lower. However, Algorithm 3 can filter the power traces for a successful fault injection and a correct ciphertext output.

2.7.2 Related Work

Masked implementations are a popular countermeasure for hardware and software implementations alike. A significant effort has been devoted to the development of higher-order masking techniques, which use multiple secret shares for each sensitive variable [77]. These techniques require a significant amount of randomness. The best schemes for masked hardware implementations with d shares (i.e. order-d implementations) use $d(d+1)/2$ random bits per data bit [70]. The best software implementations require $(d+1)$ random bits [130]. In addition, random masks have to be refreshed regularly, and non-linear operations (such

as AND) require extra masks [144]. However, none of these research efforts considers the risks introduced by requiring fresh random numbers for every iteration of a cryptographic algorithm. We observe that higher-order countermeasures require a larger amount of randomness, but otherwise they are vulnerable to our attack for the same reason as first-order countermeasures.

Over the past decade, the interaction of fault analysis and side-channel analysis has been investigated by multiple authors. It is known that a masking side-channel countermeasure is not resistant against fault analysis [11, 26]. When fault injection is used to introduce a known value (stuck-at fault), the effect of randomization can be disabled within the cryptographic algorithm. The same fault attack principle was also demonstrated on algorithms with built-in fault countermeasures and side-channel countermeasures [131]. All of this work, however, builds algorithm-specific attacks. For (differential) fault attacks, this requires insight into the fault propagation within the algorithm; for attacks on countermeasures, this requires insight into the internal design of the countermeasure. In this chapter, we perform a macro-level analysis of masking, and we point out the risks of generating and provisioning of random numbers to masked algorithms.

2.7.3 Possible countermeasures

In the proposed attack, the fault injection targets the data transfer operation between the PRNG and masked AES. The PRNG and AES themselves work correctly even if fault injection is successful. Therefore, strengthening PRNG and AES with redundancy-based fault tolerance [19, 117, 132], detection [19], and infection [117] techniques does not mitigate the attack.

To protect the data transfer operation, a software designer can replace each instruction of

this part with a sequence of instructions such that the code works correctly even if a single instruction is skipped [111]. Similarly, a designer can employ dedicated counters to keep track of the control flow of the data transfer part [92]. However, these software countermeasures bring significant performance overhead and they are not efficient against adaptive adversaries and microarchitecture-aware fault attacks [44].

2.8 Conclusion

In this Chapter, we propose a novel methodology to attack the masking countermeasure. The proposed methodology combines fault injection and first-order side channel analysis. In a masking implementation, the transfer of random masks from the PRNG to the masked cipher is an Achilles heel. Our methodology bypass the transfer operation with fault injection to make the masked cipher vulnerable to first-order side-channel analysis. We experimentally demonstrated the methodology on two masked AES implementations. Our work concludes that a secure random number transfer mechanism is required for secure masking implementations.

Chapter 3

A Low-cost Function Call Protection Mechanism Against Instruction Skip Fault Attacks

In the previous chapter, we demonstrate that function call is a weak link in the masking countermeasure; it can be easily bypassed by fault attack and remove the randomness introduced by the random number generator. In this chapter, we present a lightweight solution to protect function calls from instruction skip fault attacks. This work has been published in Proceedings of the 2018 Workshop on Attacks and Solutions in Hardware Security (ASHES) [152]

3.1 Introduction

Fault Injection attack was introduced in 1997 when Boneh et al. [25] first implemented a fault attack on a cryptographic microcontroller with a public-key cryptosystem. In a fault attack, the attacker injects well-targeted faults into the execution of the program and analyzes the response of the program to fault injection. After the attack and subsequent analysis, the adversary can retrieve the secret data processed by the electronic devices.

In a fault attack, the characteristic of a fault injected is defined by the fault model. In this chapter, we consider the instruction skip fault model [112] and propose a light-weight countermeasure scheme that can protect function calls against instruction skips. Instruction skip is a fault model where the adversary skips the execution of one or more software instructions running on embedded processors. Some cases when these instructions are replaced by other instructions which have the same effect as a **nop** instruction are also considered as a generalized versions of instruction skip fault [113].

Fault attacks based on instruction skip are simple but powerful.The adversary can utilize instruction skip to bypass security checks, change the control flow of the program or skip the security sensitive instructions in crypto-algorithms to retrieve the secret data. In an attack on RSA [83], the error-checking routine was skipped to break the fault countermeasures for RSA with Chinese Remainder Theorem (CRT). In a similar fault attack, Schmidt et al. [133] implemented an instruction skip attack on the square and multiply operation in RSA. They retrieve the RSA key by analyzing the faulty output from skipping the square operation. In the attack presented by Breier et al. [27], instruction skip is induced by laser injection on an 8-bit Atmel ATmega328P micro-controller to skip the **xor** instruction of the last round of AES, which is the *Addroundkey* operation. The last-round key is successfully retrieved by xoring the faulty ciphertext output with the correct ciphertext output. In a recent fault attack, Timmers et al. implement an instruction skip attack for privilege-escalation in the Linux OS [142].

3.1.1 Motivation

Function calls including system calls are particularly important for the integration of security components in a crypto-system. However, function calls are vulnerable to instruction skip

fault attacks. In the recent attack presented by Yao et al. [153], masking countermeasure is broken by injecting a well-targeted fault to skip the random number transfer function call, `memmove()`, at the end of random number generation. The paper points out that in the implementations of random number generators in open-source cryptographic libraries, random numbers are first generated and stored in an internal buffer and transferred to an external buffer that can be directly accessed by the user through system function calls like `memmove()` and `memcpy()`. In their attack, the mask is disabled by skipping the `memmove()` function call.

Moreover, other cipher implementations in crypto-libraries also contain sensitive functions calls which if skipped will destroy the security of the whole cipher-system. For example, in the AES-CBC mode implementation of OpenSSL, `memcpy()` function is called to update the vector for next block to current block's ciphertext output. The adversary can inject a fault and skip the `memcpy()` function call to avoid updating the vector for each block. When the initial vector remains constant in AES-CBC mode, the block cipher will not be able to hide the data pattern adequately, similar to AES-ECB mode, because of the lack of data diffusion. Thus, a single instruction skip compromises the confidentiality of AES-CBC mode.

Function calls are important but weak links in the integration of security components to a cryptosystem. A function call is vulnerable to instruction skips even if the function body is already protected against fault injection. Previous work fails to address the vulnerability and develop corresponding countermeasures. Especially, among all those functions, there are functions the body of which cannot be modified, such as system calls. In this chapter, we specifically focus on function call protection.

In this chapter, we propose a software countermeasure to protect function calls against instruction skip faults. We chose software countermeasures because they are more flexible, portable to other architectures and do not require hardware modifications. Moreover, our

countermeasure is generic to all the function calls with return values. The countermeasure does not require the modification of the function body. Therefore, our countermeasure is able to protect system calls as well. We are not aware of other work that proposes countermeasures to specifically secure function calls against instruction skip fault attack.

There exists some related work on instruction skip countermeasures. Hagai Bar-el et al. [18] proposed repeating of the algorithm in the source code level, Alessandro Barenghi et al. [20] developed selective assembly level instruction duplication and triplication protection scheme specifically for AES. Nicolas Moro et al. [113] proposed similar assembly level instruction duplication which can be applied to a larger scale of instructions but requires several transformations ahead of time. However, those countermeasures introduce significant overhead in terms of code size and execution time, and they are specific to an algorithm or an architecture. They require source code modifications on the function body. These disadvantages limit their scope of protection. Also, duplication of secure sensitive code will increase side-channel leakage[35].

3.1.2 Contributions

To address these concerns, we propose a light-weight protection scheme to protect function calls against instruction skip faults. We summarize our contribution as follows:

- We propose a light-weight software-only function call instruction skip countermeasure. This countermeasure relies on function output arguments. Defined by the ABI, a function's caller and callee always agree on a specific memory location or registers to which the return values are passed. We utilize this location information to develop instruction skip fault detection mechanism without making any changes to the function body.

- Our methodology is independent of the architecture, operating system, and programming language. Our countermeasure can be applied to any function calls with return values.
- We demonstrate the feasibility of our countermeasure and simulate the instruction skip fault on a SPARC architecture.
- To the best of our knowledge, we are the first to propose an instruction skip countermeasure to specifically protect function calls.

The remainder of the chapter is organized as follows. In the next section, we discuss the necessary background for the proposed countermeasure. Section 3 provides a brief overview of our countermeasure followed by a detailed description of our methodology. Section 4 introduces our fault simulation experimental setup and experimental results. Section 5 elaborates on discussions regarding our countermeasure and potential future work. We then conclude the chapter.

3.2 Background

3.2.1 Fault detection principles

In this section, we introduce the fault detection techniques used in fault countermeasures. Fault detection techniques are generally classified in two categories: timing redundancy and information redundancy based fault detection techniques.

The idea of timing redundancy based fault detection is straightforward. It executes the same program block twice and checks the consistency of the results from the two computations. For

example, in countermeasure proposed by Karri et al. [78], first, the input data is encrypted followed by decryption. Then, the decrypted output is compared with the original input data to detect faults. In Maistri et al.'s paper[97], the same data is encrypted twice in one function block and the result is then compared to detect anomalies. However, such timing redundancy based protection schemes either double the hardware area or double the execution time. Also, the fault may not be detected if both computations are affected by the fault. Several advanced variations of timing-redundancy based fault detection techniques have been proposed. For example, invariance in cryptographic algorithm was used to cause an imbalance in the two redundant computations[71]. Even though faults can influence both of the computations at the same time, the results will still not match. However, this protection scheme is algorithm specific, it is not generic to all the other functions.

In the information redundancy based fault detection, additional parity bit or variables are used to keep track of the execution status. In the paper presented by Lalande et al.[92], nested counters are used in every function or code block at the source code level. In their approach, the counter was incremented after each C statement and then compared with the expected value before the next statement is executed. These counters ensure the control flow integrity by detecting instruction skips. However, this approach is based on insertion of additional instructions before and after each original instruction inside the target function or code block; it requires modification of the target function and introduces a large overhead to fully protect all the sensitive functions in a program.

In our fault detection mechanism, we are not using redundancy to detect faults. Instead, we directly implement fault sensing.

3.2.2 Application Binary Interfaces

Application binary interface (ABI) is a hardware dependent contract between low-level binary code and the processor hardware. The ABI defines the calling conventions for functions, which are methods for passing function arguments and retrieving return values.

Defined by the ABI, a function's caller and callee always agree on a specific memory location or registers to which the return values are passed. We utilize this location information to develop an instruction skip fault detection mechanism without making any changes to the internal function.

We enumerate several architecture's calling conventions for processing the return values:

1) RISC-V [147] : In the RISC-V calling convention, the integer registers **a0** and **a1**, and floating-point registers **fa0** and **fa1** are used to handle the return values. In case the return values are smaller than two pointer words, there are two possible scenarios. First, if the return values are integers, they are always put in the integer registers: **a0** and **a1**. Second, if the return values are floating-point primitives or part of a structure, the return values are always held by the floating point registers: **fa0** and **fa1**. In the other case, when the return values are larger than two-pointer words, they will be passed into a pre-defined memory region, a pointer to this memory region is passed as an implicit first parameter by the function caller.

2) ARM [74]: In the ARM calling convention, for a fundamental data type which is smaller than 4 bytes, the return values will be passed to the registers starting from r0. For a data type larger than 4 bytes, similar to RISC-V calling convention, the return values will be stored in a memory and the address will be passed as an extra argument when the function is called.

3) LEON3[76]: LEON3, a 32 bit processor based SPARC architecture, is built upon register window to handle the function calls. In the SPARC calling convention, 32 registers are visible to the program and 24 of these registers are arranged in the register window mechanism. The 24 registers are equally divided into 3 groups:*out*, *local* and *in* registers.

Specifically, the *in* and *out* registers are used to pass parameters to the subroutines and receive the result. When a function is called, the caller's *out* registers will become the callee's *in* registers. In this way, the arguments are passed to the function through these registers. At the end of the function call, the callee's *in* registers will become caller's *out* registers thus pass the return value from callee to the caller.

In the SPARC calling convention, the return values are handled in three cases depending on their types: integer scalar value, floating-point value and aggregate value. In the integer scalar case, the return values are always put in the subroutine's *in* registers (which is the caller's *out* registers), starting from register **%i0** (which is caller's register **%o0**). In the floating-point scalar case, the return values are always passed through the floating-point registers, starting with the register **%f0**. In the case of aggregate return values, such as a structure in C language, the function's caller will allocate an area of memory to store the aggregate return value and the address of this memory area is stored on a fixed location on the stack.

In every ABI, the return value is always passed to a specific memory location or registers defined by the ABI. In this chapter, we utilize this feature to develop a function call protection wrapper which is generic to all function calls with return values.

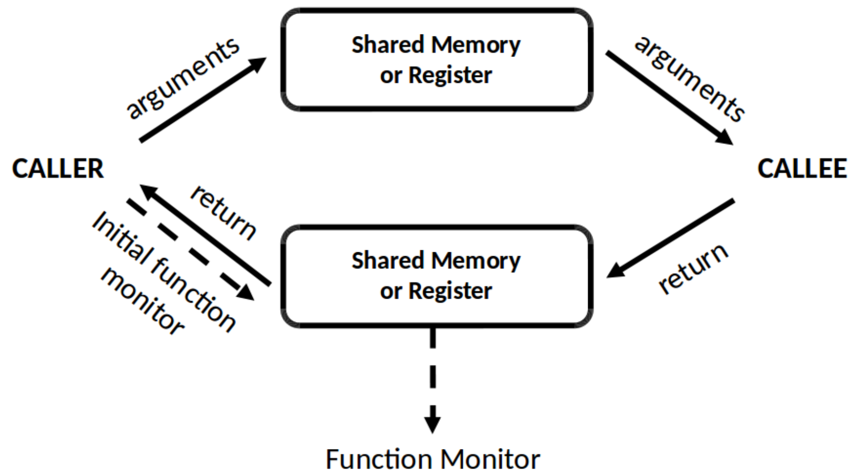

Figure 3.1: Fundamental Principles of Proposed Countermeasure

3.3 Countermeasures

This section explains our methodology for a low-cost function call protection scheme against instruction skip fault attack.

3.3.1 Fundamental Principles

In a basic function call procedure, as shown in Figure 3.1, the caller first passes the arguments to the callee using shared memory locations or registers. After the execution of the function, the callee writes the return value to the designated location from which the caller will read. In this chapter, we define any registers or shared memory locations which will be affected by the function execution as a **Function Monitor (FM)**. Listing 3.1 shows an example of an FM. The function `sqr()` updates the value of variable **b**, thus in this case the variable **b** is the FM for the function `sqr()`.

The fundamental idea of our proposed countermeasure is to initialize the FM to an unlikely

value before the function call and check if the FM has been updated after the function call. It's like putting a nut on a train rail, if the nut is broken, we know the train has passed. In our mechanism, the train is the function and the nut is the FM.

We apply this protection mechanism to standard functions with return values and we implement it as a fault detection wrapper around those functions.

There is a trade-off between fault detection rate and overhead of the detection wrapper. There is a trade-off between the correctness of fault detection and the overhead the wrapper will introduce to set up the FM and to detect changes to the FM. This trade-off depends on the quantity of the information of the target function which the proposed fault detection wrapper will use.

```
1  #include<stdio.h>
2  int sqr ( int x );
3  // main function
4  void main( )
5  {
6      int a, b ;
7      // function call
8      b = sqr ( a ) ;
9  }
10 // function definition
11 int sqr ( int x )
12 {
13     int s ;
14     s = x * x ;
15     return s ;
16 }
```

Listing 3.1: Example Code for FM

We classify our methodology in three levels, in the ascending order of the amount of the function information that the fault detection wrapper utilizes. The three levels are: ABI specific, header specific, and semantic specific level.

As shown in Figure 3.2, in the ABI specific level, fault detection wrapper only utilizes the ABI information. The targeted function is treated as a black box. In the header specific level,

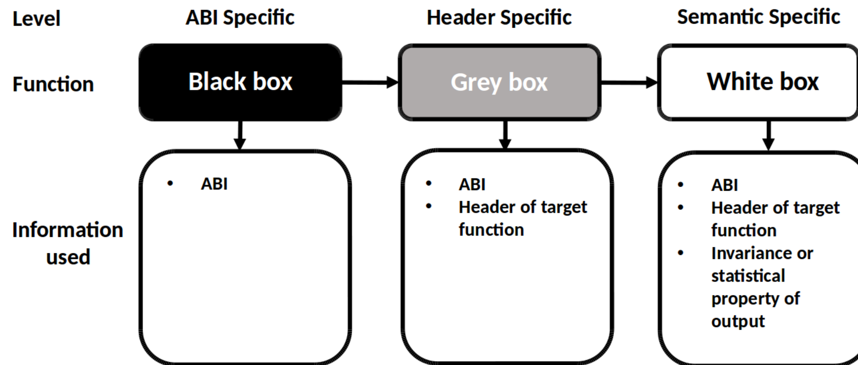

Figure 3.2: Three Levels in Methodology

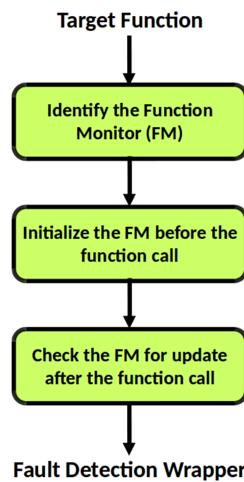

Figure 3.3: Methodology

the wrapper utilizes the function's header information. The header information includes input arguments and the user- defined registers or shared memory locations which hold the function return value. In the header specific level, the target function is treated as a gray box. In the third level, which is the semantic specific level, the fault detection wrapper further utilizes the invariance or the statistical properties of the function output. In this case, the target function is treated as a white box.

Table 3.1: Unlikely Value Example

Function API	Return Value	Unlikely Value
`void *memcpy (void*dest, const void *src, size_t n);`	a pointer to dest	value different from dest
`void *malloc (size_t size);`	a pointer to the allocated memory from the heap	top of the stack address
`char *strcpy (char*dest, const char *src);`	a pointer to the destination string dest	value different from dest
`int printf (const char *format, ...);`	the number of characters printed	-1

3.3.2 Methodology

In this section, we give an overview of the protection methodology and explain the steps involved in developing the instruction skip protection wrapper as shown in Figure 3.3.

1) Identify the FM: The first step is to analyze the FM. By analyzing the source code, we first identify the FM of the target function.

2) Initialize the FM before the function call: After identifying the FM, the second step is to initialize the FM to an unlikely value before the function call. The unlikely value can be any value with a specific pattern or a constant which statistically has a low possibility to occur based on the function. Table 3.1 shows a list of example functions and its unlikely return value. For example, let's take `malloc()` function. Based on the information from the Linux manual page, `malloc()` returns a pointer to the allocated memory from the heap. Therefore, the unlikely return value can be the address of top of the stack. The selection of the unlikely values of the FM will influence the false positive rate of the instruction skip detection. The false positive rate can be reduced by using unlikely value which has a low possibility of occurrence.

3) Check the FM for updates after function call: In the third step, the fault detection wrapper checks whether the FM has been updated from the initial unlikely value after the function call. If the function call is skipped under a fault attack, the FM will be unchanged. Therefore, the protection wrapper will compare the FM value with its initial unlikely value after the function call. If the value is unchanged, the wrapper will raise an

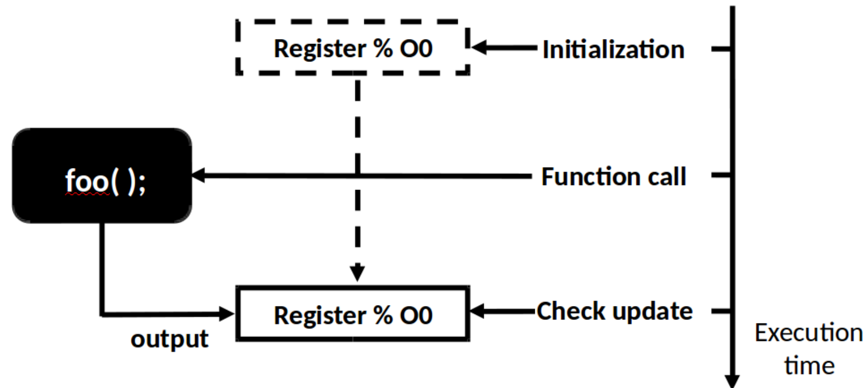

Figure 3.4: ABI specific fault detection wrapper structure

alarm signal which in turn warns that a function call skip has been detected. Moreover, the protection wrapper should be fault tolerant.

3.3.3 ABI specific Test

In the ABI specific test, the fault detection wrapper only utilizes the information about the locations of the return values of a function call. The FM in ABI specific level is ABI defined registers or shared memory locations which hold the return value. The fault detection wrapper which is developed based on our methodology can be directly applied to all the function calls with return values.

We demonstrate our methodology in the SPARC architecture using C language. The return value is passed in two forms: return by value and return by reference (pointer).

Functions Return by Value:

At the end of a function call, the output of the function is the actual value in the register. Using our methodology, we developed the fault detection wrapper for the function call of

our target function `foo()` for the SPARC architecture as shown in Figure 3.4.

1) Identify the FM: In the SPARC architecture, the return value is stored in register **%o0**, which is the FM. It should be updated with the function's return value after the function call.

2) Initialize the FM before the function call: After identifying register **%o0** as the FM, we initialize **%o0** to an unlikely value before the target function call. To directly work with the registers from our software program, we use inline assembly to initialize register **%o0**. Line 2-7 in Listing 3.2 shows the part of the wrapper code to initialize the FM. The register **%o0** is initialized to 0x12345678 before the function call.

3) Check the FM for update after function call: After function execution, the third step is to check whether register **%o0** has been updated. Lines 10-21 in Listing 3.2 show the post-execution register check of the wrapper. The function output is compared with initial value, if the value is the same, a fault handling exception will be stimulated. Note that the value of register **%o0** should not be modified by the wrapper since the function return value in **%o0** may be required by the proceeding code.

Furthermore, to make the fault detection wrapper fault-tolerant, we applied software instruction skip fault countermeasure proposed by Moro et al.. In their paper, it is assumed that it is unlikely for the attacker to inject two back-to-back faults. Based on their instruction classification, the assembly instructions in our wrapper are idempotent instructions, i.e., the instructions have the same effect when executing once or several times [113]. Therefore, we duplicate the assembly instructions to make them tolerant to instruction skips. The **be** instruction (branch equal), in line 19-20, is not an idempotent instruction. However, under our implementation settings, **be** is used to branch to the fault handling procedure. When no fault is detected, duplicating **be** instruction will not influence the control flow of the

program. Usually, when a function skip fault is detected, the fault handling procedure resets the processor and restarts the program or halts the program and raises an alarm signal. The second duplication of **be** will not be executed, thus it will not influence the control flow as well. Therefore, we can duplicate the **be** instruction in order to make our wrapper tolerant to instruction skip faults.

```
asm (
     "sethi %hi(0x12345678), %o0\n"
     "sethi %hi(0x12345678), %o0\n"
     "or %o0,%lo(0x12345678), %o0\n"
     "or %o0,%lo(0x12345678), %o0\n"
 );
//function call
  foo();
 asm (
     "mov  %o0, %g6\n"
     "mov  %o0, %g6\n"
     "sethi %hi(0x12345678), %g5\n"
     "sethi %hi(0x12345678), %g5\n"
     "or %g5,%lo(0x12345678), %g5\n"
     "or %g5,%lo(0x12345678), %g5\n"
     "cmp %g6, %g5\n"
     "cmp %g6, %g5\n"
     "be fault_handle\n"
     "be fault_handle\n"
  );
```

Listing 3.2: Protection Wrapper for functions return by value

Functions Return by Reference

In this form, the function output is a pointer pointing to the memory location which stores the actual return value. The function call will update the value of the pointer instead of updating the actual return value in register. In the SPARC architecture, the return pointer value is written into register **%o0** and passed back to the function caller through the register window. After a function call, the wrapper will check the pointer value for an update instead of the update of the actual return value.

The wrapper structure is demonstrated in Figure 3.4. The wrapper first initializes the reg-

ister **%o0** to an unlikely value and checks whether **%o0** is updated after the function call. This structure is the same as the function return by value case, so the protection wrapper in Listing 3.2 can be applied directly to the return by reference case as well. This protection wrapper is generic and can be applied to all the function calls with return value under the same ABI.

There is one corner case when the processor utilizes exactly the same registers or memory locations to pass in the function arguments and pass out the function return values. In this case, the protection wrapper can take the input arguments' values as the unlikely values and check the registers or memory locations for updates after the function call. This testing mechanism may cause false positive in one special case when the function's expected return values are the same as the input arguments' values .

3.3.4 Header Specific Test

In the header specific test, the protection wrapper employs the header information of the target function. The header information includes the function input arguments and user defined registers or shared memory location which holds the function return value. The FM in this level is the user-defined registers or the shared memory locations. Based on the number of the function return values, we divide the functions in this level into two cases: function returning a scalar and function returning a buffer.

Functions Return a Scalar

The function returning a single value to a user-defined location. Listing 3.3 shows an example of function returning a scalar. This program calls the function `foo()` (API: `int foo(int`

arg1, int arg2)) defined in the header file LibraryHeader.h. The function return value is assigned to **b**. The protection wrapper directly tests the user-defined location.

There are two testing strategies. First, the protection wrapper utilizes the function header information and tests the value of **b**. In this case, the wrapper will be developed in C or other high level programming language. Second, protection wrapper still only uses the ABI information which is the same wrapper developed in the ABI specific level. In this strategy, the wrapper is developed in assembly instructions. Moreover, since after compiling, one instruction in C can be interpreted into several assembly instructions, the protection wrapper in C can result in a larger overhead compared to an assembly wrapper.

```
1 #include<stdio.h>
2 #include "LibraryHeader.h"
3 void main( )
4 {
5     int argument_1, argument_2 ;
6     // function call
7     b = foo( argument1, argument2 ) ;
8 }
```

Listing 3.3: Example Code for functions return a scalar

Function Returning a Buffer

When the function returns multiple values, to allocate a memory area or registers for the return values. During the function call, the user-defined buffer is passed into the function as an pointer. After the execution, the function fills the buffer with return values and output the buffer. For example, in `foo()` function (API: `void foo (uint8_t *buffer, int size)`), the user-defined buffer is passed into the function and will be updated by the function with its return values. The wrapper will test the update of every value in the buffer. Figure 3.5 demonstrates the wrapper development procedure for the header specific level. We demonstrate the development procedure with `foo()` function as the target function.

Figure 3.5: Header specific fault detection wrapper structure

1) Identify the FM: The user-defined buffer will be updated after the function call. Therefore, FM for the targeted function is the user-defined buffer.

2) Initialize the FM before the function call: In this step, we need to initialize every value in the buffer to an unlikely value before `foo()` is called. As shown in the Figure 3.5, the user defines a buffer with SIZE_1 and initializes all the values to unlikely values. Afterwards, the buffer is passed as a pointer to `foo()` as input.

3) Check the FM for update after the function call: After the function call, the `foo()` function will update SIZE_2 number of values in the user-defined buffer. SIZE_2 can be equal to or smaller than the user-defined buffer size SIZE_1. When the SIZE_2 is smaller than SIZE_1, it means the function only partially updated the buffer.

In this step, the protection wrapper will check the buffer for updates after the function call.

There are two testing strategies for post-function call update checking: optimistic test and pessimistic test. For the optimistic test, the wrapper checks whether there is at least one value different from initial value. Note that the optimistic testing strategy can cause false negatives, i.e., a fault may cause a partial buffer update, but the wrapper does not detect it.

For the pessimistic test, the wrapper checks whether every value is updated from the initial value. In this case, the pessimistic test strategy has a risk of false positive, i.e., the wrapper detects the fault, but there is no real fault because the function only updated the buffer partially.

To make the protection wrapper instruction skip fault tolerant, we applied software programming patterns proposed in [150]. To prevent the attacker from bypassing for-loop check or terminate the loop early by instruction skip, we added an extra check to verify if the

Figure 3.6: Semantic specific fault detection wrapper structure

current loop counter equals the loop bound after each for-loop execution. Also, based on the assumption that it is very hard for an attacker to inject two back to back faults, instruction duplication is applied for sensitive condition checks and sensitive instructions.

3.3.5 Semantic Specific Test

In the semantic specific level, the fault detection wrapper utilizes the statistical properties of the return value. Based on this information, the wrapper can further improve FM's fault detection rate. Figure 3.6 shows the protection wrapper structure for semantic specific test.

We take random number generator function call `RNG()` (API: `int RNG(uint8_t *buffer, int size)`) as an example to demonstrate our proposed semantic specific test.

For random number function call, FM is the user-defined random number buffer. Under the semantic specific level assumption, the wrapper implements randomness tests based on the random number's statistical property after the function call, i.e. the randomness property of the value in the buffer is tested after the function call.

We develop a lightweight randomness test that can be used after the random number function call. For random numbers to be truly random, the number of ones and zeros should be approximately the same. Thus, our test checks whether the number of ones in the buffer is around half of the total number of bits (BIT_SIZE). If the result is within the range of (0.3*BIT_SIZE, 0.7*BIT_SIZE), we accept the result as 'random'. The wrapper can widen or shrink this range to change the strictness of the test. If the range is narrow, there is a higher risk of false negative, and if the range is wider, then there is a higher risk of false positive. Moreover, we implement the algorithm introduced in [12], which counts the number

of ones in parallel, to reduce the overhead in counting the number of ones.

To design a fault tolerant wrapper, we also implement sensitive condition double-checks and loop-checks as used in [150].

3.4 Results

In this section, we cover our protection wrapper overhead and experimental result for fault detection.

3.4.1 Experimental Setup

Target Function

We implement the ABI specific level, header specific level and semantic specific level of fault detection wrappers respectively on two target functions: Random Number Generator (`RNG()`) (API: `int RNG(uint8_t *buffer, int size)`) and `memmove()` (API: `void *memmove(void *dest, const void *src, size_t n)`).

In the ABI specific test, as demonstrated in section 3.3, we implement the fault detection wrapper which checks the return value in register `%o0`.

In the header specific test, the fault detection wrapper is developed based on an optimistic testing strategy for which the fault detection wrapper will declare that no fault is detected if at least one of the return values is updated after the function call. The fault detection wrapper initializes every value in the user-defined buffer to an unlikely value before the function call and it checks a select element in the buffer for the update.

In the semantic specific test, the fault detection wrapper varies depending on the target function's invariance or statistical properties. For `RNG()`, as demonstrated in section 3.5, we implement the wrapper which checks the randomness of the return value after the function call. For the `memmove()` function call protection, we utilize the invariance of the return value. Since the `memmove()` copies n bytes from memory area `src` to memory area `dest`, the value of the first elements in dest should be the same as the value of the first element in src. Therefore, the fault detection wrapper compares the value dest's first element with src's after the function call. There are also other invariants that can be used for `memmove()`, such as the invariant that the sum of all the elements in the dest is the same as the sum of src after the function call.

Fault Simulation

We simulate our software protection wrapper on LEON3 which is a 32-bit SPARC processor. We use the Gaisler's LEON3 TSIM simulator to inject faults and skip instructions. We wrote a script to interface with TSIM simulator that automates the instruction skips. It enabled single instruction skips in a sequence within a user-defined instruction range. We define an instruction skip simulation window around the function call instruction. Practical fault injection is imprecise and shows ambiguity with respect to the target instruction. During the fault injection, the adjacent instructions around target instructions may also be influenced. The window is a fault injection range around the target function call instruction. The size of the window reflects the accuracy of fault injection devices. It depends on the fault target processor's speed and fault injection devices resolution.

In our simulation, we set the window size to 11. Therefore, the window is starting from the 5th instruction before the target function call instruction and ends at the 5th instruction after the target function call instruction. We simulate instruction skip for each of the 11

instructions within the simulation window, and simultaneously monitor the function output and the wrapper detection performance.

3.4.2 Fault Detection Performance

Under the instruction skip fault injection, there are three possible outputs: faulty output, correct output, processor crash or timeout. As shown in the Table 3.2, the response of the fault detection wrapper has 5 cases:

1. X_0 **- False Positive**: Output is correct, the wrapper detect fault;
2. X_1 **- True Positive**: Output is faulty, the wrapper detects fault;
3. X_2 **- True Negative**: Output is correct, the wrapper does not detect fault;
4. X_3 **- False Negative**: Output is faulty, the wrapper does not detect fault;
5. X_4 **- Time Out**: If the program crashes as a result of the skip, no output is produced.

We calculate 3 metrics: Correct Response Rate (CRR), Fault Detection Rate (FDR) and Fault Detection Quality(FDQ), to quantify the fault detection performance,i.e., the security level of our wrapper.

$$CRR = \frac{S_{CorrectResponse}}{S_{window}} = \frac{X_1 + X_2}{\sum_{i=0}^{4} X_i} \tag{1}$$

$$FDR = \frac{F_{detected}}{F_{total}} = \frac{X_1}{X_1 + X_3} \tag{2}$$

$$FDQ = CRR + FDR \quad (3)$$

Equation(1) defines CRR. Referring to the cases shown in Table 3.2, $S_{CorrectResponse}$ is the total number of correct responses $(X_1 + X_2)$ from the wrapper, and S_{window} is the window size which makes the total number of cases. The CRR reflects the overall fault detection performance including false positive and false negatives. Ideally CRR is 1 which means the fault detection mechanism correctly responds to all injections.

In the Equation (2), F_{total} is the total number of faults that can be observed from output. $F_{detected}$ is number of faults that our wrapper detects. The FDR captures how good the fault detection techniques are able to detect true faults, and ideally FDR is 1 when all the true faults are detected.

As defined in Equation(3), FDQ is the sum of CRR and FDR, FDQ reflects the overall quality of the fault implementation. Ideal value for the FDQ is 2.

Table 3.3 and 3.4 show the experiment result for `RNG` function and `memmove()` function fault detection wrapper respectively. Table 3.4 shows the fault detection result for three different levels of protection wrapper for target function `memmove()`. In the ABI specific test, out of the 11 instruction skips within the fault injection window, there are 0 cases when the outputs were correct and the wrapper detects fault $(X0)$, 2 cases when the outputs were faulty and the wrapper detects fault $(X0)$, 2 cases when the outputs are faulty and wrapper does not detect fault $(X2)$, 3 cases when the outputs are faulty and wrapper does not detect fault $(X3)$, and 4 cases when the instruction skip cause the processor crashes or program

Table 3.2: Fault Detection Wrapper Response Cases

Alarm \ **Output**	**Correct**	**Faulty**	**Crash or Time Out**
Yes	X_0	X_1	X_4
No	X_2	X_3	

Table 3.3: Test Results for RNG( ) Function Call

Result \ **Test level**	**ABI Specific**	**Header Specific**	**Semantic Specific**
$X0$	0	0	0
$X1$	1	1	2
$X2$	8	7	7
$X3$	1	0	0
$X4$	1	3	2
Correct Response Rate(CRR)	0.82	0.73	0.82
Fault Detection Rate (FDR)	0.73	1.00	1.00
Fault Detection Quality (FDQ)	1.55	1.73	1.82

execution timeout. So for the ABI specific test, CRR is the sum of $X1$ and $X2$ cases which is 4 and divided by the window size 11. Thus the CRR value is 0.36. FDR is 2 ($X1$) divided by total number of faulty cases which is 5, thus the FDR is 0.4. In total, FDQ for ABI specific test is 0.76. The same analysis is applicable to the result for header specific test and semantic specific test. As can be observed from FDQ shown in the last row in Table 3.4, From the most simple ABI specific level to the complex semantic specific level, the FDQ increases, which indicates increasing security level of the fault detection wrapper. We can observe similar behavior for the experiment result for target function `RNG()` shown in the Table 3.3.

Table 3.4: Test Results for memmove( ) Function Call

Test level / Result	ABI Specific	Header Specific	Semantic Specific
$X0$	0	0	0
$X1$	2	3	4
$X2$	2	2	2
$X3$	3	2	1
$X4$	4	4	4
Correct Response Rate(CRR)	0.36	0.46	0.55
Fault Detection Rate (FDR)	0.40	0.60	0.80
Fault Detection Quality (FDQ)	0.76	1.06	1.35

Table 3.5: Overhead in Code Size

Test level / Function	ABI Specific (Bytes)	Header Specific (Bytes)	Semantic Specific (Bytes)
RNG( )	100	400	1128
memmove( )	115	208	352

Table 3.6: Overhead in Cycle Count

Test level / Function	ABI Specific (cycles)	Header Specific (cycles)	Semantic Specific (cycles)
RNG( )	20	99	838
memmove( )	24	64	86

Table 3.7: Estimated Overhead of Related Work

Overhead / Function	Execution Time (cycles)	Duplication Overhead (cycles)	Counter [92] Overhead[01] (cycles)
RNG( )	7993	7993	31972
memmove( )	133	133	N/A

3.4.3 Overhead

Table 3.5 and Table 3.6 show the overhead in code size as well as cycle count incurred by our wrapper over the non-protected version. In the protection wrapper of `RNG()`, for the most simple ABI specific level, only 100 bytes in code size and 20 cycles in performance overhead is introduced by the wrapper. The overhead increases depending on the level of security from ABI specific test to semantic specific test.

The result from Table 3.4- 3.6 indicate the trade-off between the security level of the fault detection wrapper and the overhead introduced. For example, in the `memmove()` function call, from ABI specific test to semantic specific test, the wrapper's FDR increases which indicates the improvement of overall fault detection performance. As a trade-off, the overhead increases simultaneously as the fault detection performance improved. This is because more information from the target function is used in the fault detection wrapper to make the fault detection more comprehensive.

3.5 Discussion

In our protection mechanism, the overhead for ABI specific test is approximately stable for different target functions since the ABI specific wrapper always checks the update of ABI defined return registers or shared memory for updates. In the header specific test,

the overhead varies depending on the user-defined buffer size. For the semantic level test, the overhead depending on the statistical property or the invariance of the target function utilized by the fault detection wrapper.

There is a trade-off between the security level of the fault detection wrapper and the overhead introduced. Depending on the security requirement of protection target security system, we can choose which level of fault detection wrapper to implement.

Our protection mechanism specifically targets the protection of function call against an instruction skip fault. It is relatively a low-cost approach to protect function call fault against instruction skips. The most common way to protect the function call is function call duplication, i.e, execute the function twice. However, the performance overhead introduced by function call duplication is the target function execution time and it varies depending on different target functions. Table 3.7 shows the estimated overhead for duplication countermeasure and countermeasure introduced by Lalande et.al [92] which utilizes nested counter as we introduce previously in section 2.1. The `memmove()` function call takes 133 cycles to execute when the `memmove()` size is 3. If utilizing the function call duplication, the performance overhead will be approximately 133 cycles. For the counter countermeasure which requires modification on the function body, it is not applicable to system calls. However, as shown in the Table 3.6, the most expensive overhead for `memmove()` protection which is 86 cycles is still smaller `memmove()`. For large functions, the advantage of our proposed mechanism is more obvious. In our `RNG()` case, the `RNG()` execution time is 7993 cycles, so the performance overhead for duplication will be approximately 7993 cycles, and overhead for the counter countermeasure is 31972 cycles (for ARM-V7). Compared to the overhead of our proposed mechanism as shown in Table 3.6, the overhead of duplication and counter countermeasure is much larger. By comparison, the overhead of our proposed mechanism is lightweight.

There is one limitation in measuring the detection rate of our proposed countermeasure. In the future work, a benchmark need to be developed to generate non-deterministic function calls in order to get the distribution of overhead for the proposed countermeasure. Significant amount of non-deterministic function call samples are needed to generate the overhead distribution, CRR and FDR.

3.6 Conclusion

In this chapter, we have introduced a software fault detection mechanism to specifically protect the function call against instruction skip fault attack. Our methodology does not require modification of internal function and is generic to all the function calls with return values. Our proposed methodology is independent of architecture, operating system, algorithm and programming language. We demonstrate our methodology on Gaisler's LEON3 simulator and we show that the performance overhead of our countermeasure is low.

Chapter 4

Verification of Power-based Side-channel Leakage through Simulation

Traditionally, side-channel leakage evaluation after the chip tape-out (post-silicon). Once the leakage is detected, it is already too late to make any mitigation. In this chapter and the next chapter, we demonstrate our contributions in pre-silicon side-channel leakage evaluation. In this chapter, we build up simulation-based leakage detection and evaluate the trade-offs for simulation abstract level in terms of capturing the side-channel leakage. This work has been published in 2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS) [155].

4.1 Introduction

Power-based side-channel leakage occurs when circuits handle secret data. An adversary who can measure these data-dependent power variations can mount a side-channel attack, which reconstructs the secret data from a collection of side-channel leakage measurements [52]. In recent years, the risk of side-channel leakage has risen to prominence, together with the proliferation of information technology in embedded, mobile and portable form. Success-

ful side-channel attacks have been demonstrated on a wide range of commercial technology ranging from FPGA bitstream protection to wireless car keys. As a result, there is a pressing need for designers to produce designs which are free from side-channel leakage vulnerability. In contemporary practice, side-channel leakage is tested by measuring a prototype implementation. This task, often carried out by a specialized testing lab, is time-consuming and expensive. To reduce cost and to speed up the design cycle, we need a methodology for *side-channel leakage verification*, which tests for the presence of side-channel leakage vulnerability from the design files, and which can be completed pre-tapeout. Hence we need accurate modeling of side-channel leakage. We need high-resolution power estimation to understand the time-dependent power consumption of individual components in a design. There is a trade-off between the level of detail in time-dependent power modeling, and the accuracy of side-channel leakage prediction before tape-out. Intuitively, highly detailed power models will be better at predicting side-channel leakage, but they will be more expensive in simulation cost.

In this contribution, we describe the process of side-channel leakage verification (SCLV). We first distinguish traditional side-channel leakage assessment (SCLA) and SCLV. SCLA works on physical measurements from a design prototype. SCLA tests the hypothesis whether side-channel leakage is present or not. SCLV, on the other hand, uses the design description to simulate power measurements before a physical prototype is available. SCLV can explain the cause of side-channel leakage, and it can help a designer to fix the vulnerability of a design before tape-out.

The remainder of the chapter is organized as follows. In Section II, we review known causes of power-based side-channel leakage. In Section III, we introduce SCLV. Our main contribution is to link the different simulation abstraction levels to specific circuit effects known for side-channel leakage. In Section IV, we present experimental results comparing physical

measurements on a chip prototype with power simulations. We then conclude this chapter.

4.2 Power-based Side-channel Leakage

Power-based side-channel leakage is the power consumption variation caused by the processing a secret value V internal to the circuit under consideration. In this section, we describe the causes of this power consumption variation. This will provide the basis for the discussion on side-channel leakage verification in the next section.

Sources of side-channel leakage Consider a digital design that works with a secret value V. The best known source of power-based side-channel leakage comes from secret state transitions caused by a secret bit V propagating on a digital net in the design. Each secret state transition ($0 \rightarrow 1$ or $1 \rightarrow 0$) causes a power spike proportional to the capacitive load of the net. The power spike appears as side-channel leakage in the overall dynamic power consumption of the design. The power is dissipated by the drivers of the net. A larger net with a proportionally larger loading capacitance requires larger drivers. A larger net will therefore contribute a higher amount of side channel leakage per transition. The dynamic power consumption of registers is directly related to the (potentially secret) values stored in the register. For this reason, the dynamic power consumption of registers is a popular target for side-channel attacks. When combinational logic processes a secret value V, the dynamic power consumption follows a more complex pattern. Combinational logic suffers from glitches, short data-dependent transitions caused by the intermediate evaluation of the logic. Glitches are non-linear effects that depend on the specific value of the (secret) inputs. This property has been used to break a masking countermeasure against side-channel leakage [100]. Nowadays, glitches are treated as a liability in secure circuit design, and

designers will generally try to minimize them or else adopt a design style that avoids glitches altogether. A second source of power-based side-channel leakage occurs through the static power consumption of circuits, caused by the static (constant) value of a secret value on a net. The dependency of the static power on the input values on a gate enables a successful side-channel attack [109]. The side-channel leakage from static power consumption is less pronounced than the leakage from dynamic power consumption, but it cannot be ignored.

There are secondary sources of power-based side-channel leakage, as well. The circuit power consumption is affected by parasitics including layout capacitance and cross-coupling capacitance. This leads to layout-dependent side-channel leakage [48]. Cross-coupling is a non-linear effect that can break masking countermeasures [47]. Another secondary factor in power-based side-channel leakage is the layout asymmetry and process manufacturing variations from one circuit to the next. For example, the tiny delay variations caused by structural asymmetry may cause exploitable side-channel leakage in circuits that rely on balanced timing [55], including hiding-based side-channel countermeasures [143].

In summary, the power consumption is an undisputed source of side-channel leakage. However, there are a large number of linear and non-linear power effects that combine into the overall power consumption. Many of these effects have been shown to contain exploitable side-channel leakage. This makes side-channel leakage verification both interesting and relevant.

Noise in side-channel leakage A side-channel attack extracts an information-carrying signal from noise. Hence, noise plays a role in side-channel verification. Noise degrades the quality of side-channel leakage. Therefore, from a defensive perspective, zero noise represents a worst-case scenario. While zero noise cannot be achieved in practice, it is easy to achieve in the simulations that support side-channel leakage verification. Previous research [54]

provides a comprehensive summary of the noise sources in a side-channel attack. Physical noise originates from random phenomena such as thermal noise. Measurement noise is the external noise added by measuring the power consumption signal with imperfect, practical equipment. Algorithmic noise is the noise caused by unrelated digital activities in the design under test, such as dynamic power variations caused by the processing of non-secret bits or unrelated bits [49, 57]. Finally, modeling matching noise is the noise caused by the statistical mismatch in the hypothesis test used in the side-channel attack. Regardless of the noise source, a precise modeling of the noise is generally difficult – and perhaps from side-channel leakage verification point-of-view, it's not that critical as it does not represent a worst-case scenario.

4.3 Side-channel Leakage Verification

In this section, we explain the difference between side-channel leakage verification (SCLV) and side-channel leakage assessment (SCLA). SCLA detects side-channel leakage from measurements made on a prototype, but does not explain the cause. SCLV uses simulation to explain the cause of power-based side-channel leakage before tape-out.

Side-channel Leakage Assessment SCLA confirms the presence of side-channel leakage in a design. Because of the complex nature of side-channel leakage, SCLA is typically performed after tape-out, using power measurements on a physical prototype. There are two approaches. First, the tester can evaluate side-channel leakage by implementing actual side-channel analysis attacks (SPA, DPA, template attacks [101]). Second, the tester can use statistical tests to confirm power-based variation that *may* be related to side-channel leakage (TVLA [58, 67], χ^2-test [110]). A statistic-based assessment does not perform a side-channel

attack, but aims to distinguish the power consumption distribution of a design under two well chosen input data sets. A statistic-based assessment is therefore easier to apply, but it may lead to false positives: suspected side-channel leakage does not imply that an attack will succeed. In general, side-channel leakage assessment can only evaluate the *presence* of the side-channel leakage.

Side-channel leakage verification Like SCLA, side-channel leakage verification (SCLV) aims to confirm the presence of side-channel leakage in a design. But unlike SCLA, SCLV uses the design description to simulate power measurements rather than using physical power measurements from a prototype. This brings two advantages. First, side-channel leakage can be tested early in the design, well before tape-out. While the simulated side-channel leakage from a high-level design model in RTL may be less accurate compared to the actual measurement on a prototype, the simulation result offers the benefit of early detection of side-channel leakage design errors. It is well known and accepted in the design community that errors are harder to fix in the later stages of a design. The second advantage of SCLV is that it can explain the cause of side-channel leakage. Because power is simulated from the design description, a designer can identify the design components that are at the root of the side-channel leak. Therefore, design fixes such as the selective application of side-channel countermeasures – which would be impossible using SCLA – are a viable strategy using SCLV.

Abstraction levels Power simulation applies at all design abstraction levels: RTL, gates, and transistors. Low abstraction levels simulate a design in greater detail and are therefore capable of capturing a wider variety of power-based side-channel leakage. On the other hand, they consume more simulation time. Table 4.1 summarizes the main properties of three relevant abstraction levels – RTL, gates, transistors – and their ability to capture

Table 4.1: Abstraction Levels for Side-channel Leakage Verification

Abstraction Level	**Timing Model**	**Power Model**	**Side-channel Leakage**
RTL	clock cycle	toggle count	logic transition
Gate	discrete-event	weighted toggle	logic transition + glitches + static power
Transistor	continuous time	circuit analysis	logic transition + glitches + static power + parasitics

side-channel leakage. *Logic transition* leakage refers to side-channel leakage that is directly reflected in transitions of variables (signals and wires) in an HDL model. Classic DPA attacks typically aim for logic transition leakage. *Static power* refers to leakage from static power consumption, which can be computed from a technology-mapped netlist. *Glitches* and *parasitics* refer to secondary sources of side-channel leakage, which are a concern with most side-channel leakage countermeasures.

Related work Several recent research efforts are centered around the concept of SCLV. RTL-PSC [72] describes an SCLA technique which is performed at RTL-level and identifies side-channel leakage source at the module-level. RTL-PSC is oblivious to leakage effects that require transistor-level or gate-level modeling. Architecture Correlation Analysis (ACA) uses gate-level simulation to identify the side-channel leakage source at the level of a single cell [59]. ACA further proposes a spot fix technique to remove side-channel leakage from such cells. Karna [137] partitions a chip at layout-level and determines a TVLA leakage metric for each area in order to locate the leakage source. Karna then tries to scale down the power variations from problematic layout areas.

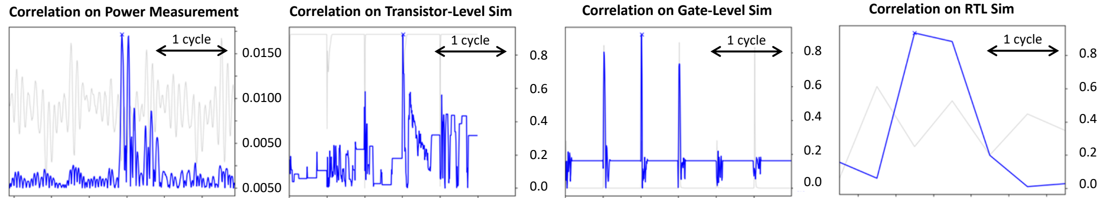

Figure 4.1: Visual comparison of power signals (grey) and correlation graphs (blue) for measurement, transistor-level simulation, gate-level simulation, and RTL simulation

4.4 Experimental Results

We demonstrate the feasibility of SCLV using simulated side-channel attacks on an SoC called FAMEv2, implemented in TSMC 180 nm standard cells. The chip contains a 32-bit on-chip core, 128 KB on-chip memory and several coprocessors. One of these coprocessors, a AES-128 cryptographic accelerator with a 1 cycle-per-round parallel architecture, is the target of the experiments. The coprocessor accepts a 128-bit key and plaintext through a memory mapped register. We have fabricated the chip and measured the power consumption with a current probe as well as with an EM probe while the chip performs encryption with the AES coprocessor. In addition, we simulated the power consumption of the AES module at RTL, gate-level and transistor-level.

Figure 4.1 is a visual comparison of the power traces (light gray) and correlation power analysis results (blue) for the measurements, transisor-level, gate-level and RTL simulation. The power trace for each case is clearly distinct and highlights specific artifacts of the selected level of abstraction. Measured power traces are noisy and distorted by parasitics. The measurements in Figure 4.1 (left) show a ringing effect, absent from the simulation. Simulated power traces are noiseless. In between up- and down-going clock edges, the simulated power drops almost to zero, while there are very sharp peaks at the clock edges. The RTL power simulation delivers only two samples per clock cycle and therefore lacks the ability to drop to zero in between clock edges.

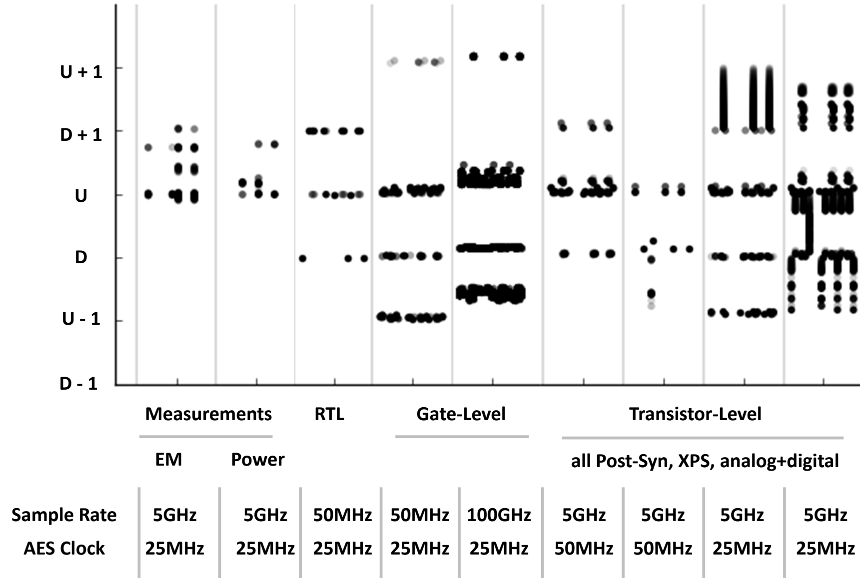

Figure 4.2: Predictive quality of power simulation for side-channel leakage detection. This plot compares the leaky points of measurements and simulations at different levels of abstraction (X) and as a function of AES clock cycle (Y).

We performed a power correlation analysis on the output of the AES state register. In SCLV the power correlation is done using a *known* key, such that any AES round can be selected for power analysis. In all cases from Figure 4.1, a correlation peak is found at the upgoing clock edge of the AES round under investigation. The height of the correlation peak of the measurements is small, about 40 times smaller than the correlation peaks found for simulation traces. The correlation on measurements also requires far more traces: 500K traces as opposed to 256 traces for simulation correlation plots. The correlation from the measured power trace persists for about half a clock cycle. On simulated correlation plots, peaks are found over several up- and down-going clock edges; we will investigate this in further detail. Overall, Figure 4.1 reveals that, while simulations can predict the position (AES clock cycle) of correlation peaks, the shape of simulated correlation peaks may differ, sometimes significantly, from measured correlation peaks.

Figure 4.2 provides a systematic comparison of the ability of (RTL, gate-level, transistor-

level) power simulation to predict side-channel leakage from a physical prototype. The SCLV is done using correlation analysis on the output of the AES MixColumns step. The Y axis of the graph represents the (U)p and (D)own edge of the clock signal over three clock cycles of investigated leakage. The X axis represents different measurements (EM and power) and simulation (RTL, gate-level and transistor-level). Each dot in the graph represents the successful recovery of a key bit, and the time instant where this happens is called the leaky point. For the selected power model, up to 32 key bits can be detected, so up to 32 dots can be plotted side-by-side at a leaky point. The number of traces used for the shown set of leaky points varies across each case. Power measurements use 500K traces, EM measurements use 75K traces and all simulations use 256 traces. A false negative in SCLV happens when a measurement picks up a leaky point that is not identified by simulation. Approximately 15% of the measured leaky points are false negatives. A false negative can occur, for example, when the physical artifact contains parastic elements not captured in the design model. A false positive in SCLV happens when a simulation picks up a leaky point that is not identified by measurement. Among false negatives and false positives, the former are more critical since they represent a failed SCLV.

The feasibility of SCLV strongly depends on the ability to simulate power traces with sufficient accuracy in a reasonable amount of time. The full FAMEv2 design contains over 120,000 cells as well as 26 SRAM arrays. Table 4.2 illustrates simulation performance as a function of simulation abstraction level. To improve simulation performance at lower abstraction level, we reduced the scope of the simulation to the module of interest (AES) and to the simulation time frame of interest (AES encryption). We replaced full analog simulation with mixed analog-digital simulation. With these optimizations, we conclude that a 24-hour turn-around simulation time at the most detailed level of abstraction is feasible for this case. However, we emphasize that SCLV can be *fully parallelized* over the traces in a simulation.

Table 4.2: Simulation Performance in function of Abstraction Level

Abstraction	**Level**	**Tool**	**Scope**	**Time/Trace (min)**
RTL	Pre-Syn	Joules	Full Chip	2.5
Gate	Post-Syn	Voltus	Full Chip	22
Transistor	Post-Syn	XPS	AES Block	7
Transistor	Post-Layout	XPS	AES Block	28
Transistor	Post-Syn	APS	AES Block	100
Transistor	Post-Layout	APS	AES Block	383

Given sufficient compute resources and, if applicable, tool licenses, the entire process can finish in the time needed to simulate a single trace at the most detailed level of abstraction.

Conclusions

We introduced side-channel leakage verification (SCLV) as a method to address side-channel leakage issues early in a design flow. SCLV fits in a methodology that trades off the security of a design, next to its power consumption, its area, and its performance. For the case of an unprotected AES design, we show that every simulation abstraction level reveals correlation-based side-channel leakage. However, we expect that the design of side-channel countermeasures will justify the use of simulation at lower abstraction levels.

Chapter 5

Pre-silicon Architecture Correlation Analysis (PACA): Identifying and Mitigating the Source of Side-channel Leakage at Gate-level

In the previous chapter, we investigate simulation-based side-channel leakage detection at the pre-silicon stage. In this chapter, we present our work in pre-silicon root cause analysis. We develop a methodology that enables the designer can precisely detect and mitigate the root cause of side-channel vulnerabilities within the design. Part of the result has been published in the 2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST) [154] and an extended version of HOST paper has been published in eprint [157].

5.1 Introduction

Power-based side-channel leakage occurs when a secure chip performs operations that depend on an internal secret value such as a cryptographic key. An adversary who observes the chip power consumption can derive the internal secret value through differential analysis

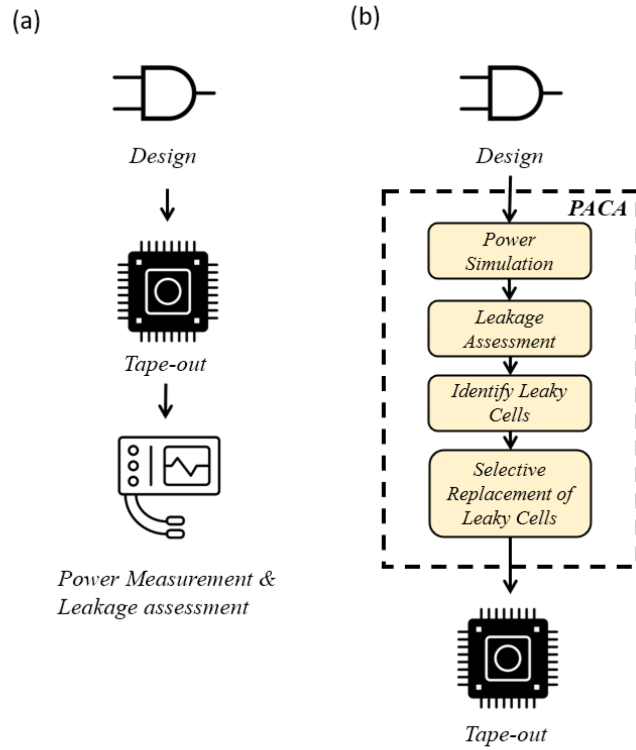

Figure 5.1: (a) Traditional side-channel leakage assessment flow. (b) Proposed PACA flow.

techniques that correlate a power model of the secret activity with the observed power consumption. In recent years, side-channel vulnerabilities have risen to prominence and successful side-channel attacks have been demonstrated on a wide range of devices from small IoT devices to large cloud computing systems. Therefore, the evaluation of the side-channel leakage has become a critical component in the electronic design flow of secure chips to avoid costly post manufacturing evaluation and reiteration of the design.

As shown in Fig. 7.1(a), the conventional method to evaluate side-channel security vulnerabilities occurs after the chip tape-out. Designers measure the prototype of the chip to assess the vulnerability. However, once a side-channel leak is confirmed, it may be too late to fix it. In the worst case, a side-channel leakage-related design mistake cannot be fixed until the next version of the chip. Another disadvantage of side-channel security evaluation by

means of a chip prototype measurement is that it is difficult to precisely locate the leakage source, especially in a complex design. Therefore, the conventional method of applying a side-channel countermeasure, such as power-randomization, hiding, or masking, is to proactively protect the whole design. However, these techniques will introduce a large overhead, with a cost proportional to the size of the module that must be protected. The overhead can be reduced by limiting the countermeasures to a small section of the chip, but then the designer must identify the precise cells which contribute to the side-channel leakage. To our knowledge, there are no tools to identify the source of side-channel leakage in a design at the granularity of a cell.

Motivated by the challenges of power-based side-channel leakage mitigation in modern chip design, we describe Pre-silicon Architecture Correlation Analysis (PACA). PACA introduces two major and novel contributions.

- PACA develops a gate-level netlist analysis methodology that enables designers to precisely identify the source of side-channel leakage in a design at the granularity of a single cell. PACA operates on the pre-silicon design description. Using experimental results from a practical SoC design, we show that only a small number of cells are significantly contributing to side-channel leakage.

- We propose **selective replacement** as a low-cost side-channel countermeasure. By protecting only the most leaky cells in a design, the overall side-channel leakage can be significantly reduced, and at a very low cost. We demonstrate selective replacement on an AES Sbox, where we replace the leaky cells identified by PACA with side-channel protected cells that use WDDL [143]and internal energy buffering [68] respectively.

Fig. 7.1(b) illustrates the proposed PACA methodology. Compared to the traditional side-channel leakage assessment, all the procedures in PACA happen before the chip tape-out.

Given a target design, PACA estimates power measurements from the DUT based using power simulation. PACA then performs side-channel leakage assessment of the design activity of interest (such as encryption). Using the side-channel assessment, PACA then ranks all the cells in a netlist with respect to their contributions to the side-channel leakage (Section 5.3). The ranking is numerically expressed using the Leakage Impact Factor (LIF). PACA then applies selective replacement to the high-LIF cells, thereby protecting the design while simultaneously reducing the overhead of side-channel countermeasures.

The structure of this chapter is as follows. The next section reviews related work in simulation-based side-channel leakage assessment. Section 5.3 describes the proposed PACA methodology. In Section 5.4, we explain and demonstrate the effectiveness of the methodology on a simple circuit. Next, we apply PACA to a SoC. We analyze an individual module as well as the impact of integrating this module in a complete SoC. Section 5.5 and Section VI discuss the PACA methodology on two encryption modules, an AES coprocessor and a PRESENT encryption module respectively. Section 5.7 shows the result of applying the methodology to the analysis of an SoC bus transfer. In section 5.9 and section 5.8, we demonstrate our proposed countermeasure concept of selective replacement. We provide several discussions about the relevant issues of PACA in section 5.10. We then conclude the chapter.

5.2 Related Work

The structure of PACA has three stages:

- Stage-I: Power simulation and leakage detection
- Stage-II: Identification of leakage source in the design

- Stage-III: Selective replacement mitigation of leakage sources

PACA's major contributions and novelty are at Stage-II and Stage-III. Most of the existing efforts, academic as well as industrial, center around pre-silicon side-channel emulation and are only focusing on power simulation and leakage detection (Stage-I). However, the problem is that even though one can detect the side-channel leakage at the pre-silicon early design stage, it is still very hard to locate problematic elements in the design and fix them. The leakage source identification (Stage-II) and leakage mitigation (Stage-III) are unique contributions by PACA.

Existing Work in Power simulation and Leakage Detection (Stage-I) Many existing works have investigated simulation techniques to simulate the side-channel effects at early design time (pre-silicon). These works present simulation methods, and exploit different aspects in simulation techniques, such as simulation accuracy, speed, and automation, to reproducing side-channel leakage and to test countermeasure at design time. **We would like to specifically distinguish PACA from those works.** ELMO models power-based side-channel leakage based on the instruction opcode and operand values [103] . Similarly, MAPS creates an ISA based simulator specifically for Cortex-M3 [37]. Instruction-based power models can capture some transition-based leakage, but they miss side-channel leakage stemming from the (potentially unknown) processor-internal effects [135]. Other techniques have also been proposed to simulate at lower abstraction level (gate-level, transistor level) in order to achieve higher simulation accuracy. One representative technique is CASCADE [136]. This work investigates power simulation at gate-level and transistor level. CASCADE is an EDA tools-based framework to automate power simulation and side-channel leakage evaluation at design-time. Similarly, Regazzoni *et al.* proposed a simulation-based methodology to evaluate the side-channel resistance of a cryptographic functional units [128]. This

work used transistor-level simulation (SPICE-level) to generate simulated power traces and apply DPA and CPA attacks. Debande *et al.* proposed profile modeling for improving the accuracy of simulation [40]. Recent developed commercial tools such as Virtualyzr by Secure IC [2] and FortifyIQ [1] also only focus on providing design-time analysis services including power simulations and side-channel evaluations on the simulated traces. Those aforementioned works targeted to build up accurate simulation models while none of these methods investigates to identify the leakage source in the design, which is the main contribution of PACA.

Existing Work in Identification and Mitigation of Leakage Source (Stage-II and Stage-III) Centering around the topic of how designers can use design data to identify the source of side-channel leakage in a design, several research works have popped up in recent years. We categorize those efforts into different abstraction levels. At the RTL-level, RTL-PSC [72] and PARAM [62] developed pre-silicon methods to simulate the power consumption and identify specific state elements that contribute to side-channel leakage. However, both works have limitations. First, the simulation accuracy is low. Since they simulate power at the register-transfer level (RTL), the internal state of the design is fully visible, but the combinational logic remains hidden in high-level expressions, and low-level effects such as glitches as well as the effects of physical placement and routing are ignored. Second, both works can only identify the leakage source to the granularity of individual design modules. At the gate-level, Karna [137] uses structural information from the layout. Karna partitions a chip spatially in a grid, and determines a TVLA leakage metric [21] for each grid cell. Karna thus identifies side-channel leakage with the spatial locality. The spatial resolution of Karna is limited by the layout area over which TVLA is computed, which may still contain many cells. A second challenge is that TVLA is not an exact leakage metric but may lead to false positives.

Another branch of the efforts to identify leakage sources related to information flow tracking. Information flow tracking techniques automatically identify causal dependencies between the different parts of a design, and therefore these techniques can analyze the dependencies between a sensitive or secret input and an observable design output. At the register-transfer level, SecVerilog analyzes hardware information flow to detect timing-based channels [159]. At the gate-level, GLIFT similarly detects timing-dependent information leaks [115]. However, information-flow-based mechanisms cannot express power-based side-channel leakage.

PACA also focuses on identifying the specific design elements that cause side-channel leakage. Compared to the aforementioned works, in terms of accuracy, PACA operates at the gate-level, which offers a good trade-off between design abstraction (simulation speed) and side-channel leakage modeling detail. In terms of leakage source granularity PACA can identify, PACA is able to narrow down the source of side-channel leakage to individual gates. This is considerably more precise than any related technique previously discussed. Because of the high resolution in root-cause identification of side-channel leakage, targeted countermeasures can be applied. As previous authors have repeatedly shown, side-channel leaks can often be attributed to a single gate [100]. We will show that the selective replacement used in PACA is both area-efficient and effective at mitigating the side-channel leakage.

5.3 PACA Methodology for Identifying the Leaky Cells

In this section, we describe the PACA methodology for identifying the Leaky Cells by computing Leakage Impact Factor for each cell in the design. The LIF is a dimensionless number that expresses the contribution of the cell's power consumption to the side-channel leakage of a design, and a higher LIF indicates a higher contribution. Fig. 5.2 demonstrates how the Leakage Impact Factor for each gate can be derived using the existing simulation design

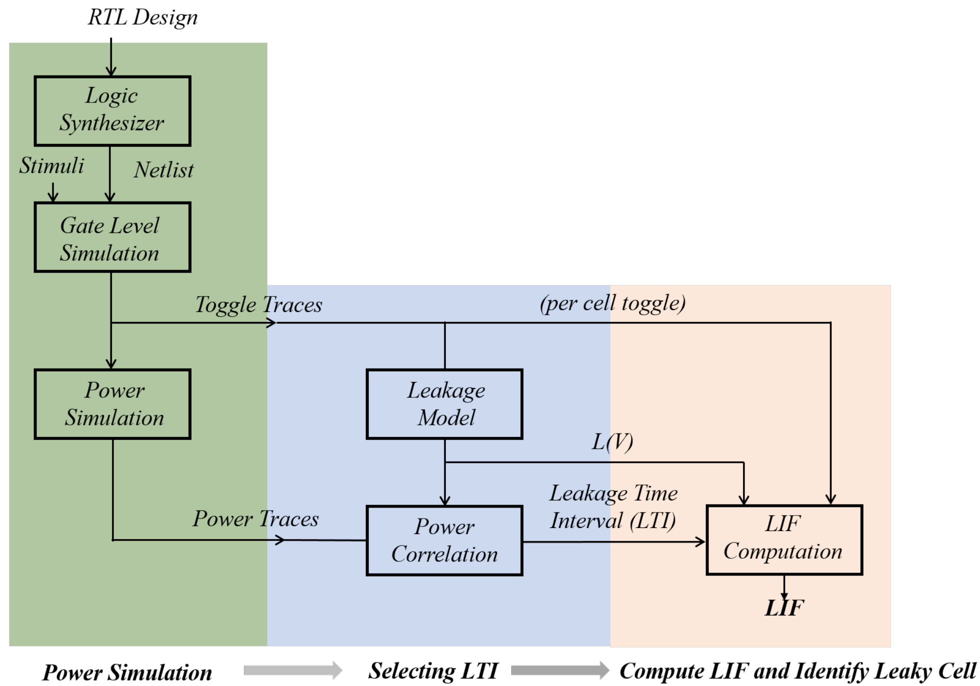

Figure 5.2: PACA flow for Identifying Leaky Cell

flow with additional postprocessing. PACA uses toggle traces as well as power traces, which are extracted from gate-level logic simulation and gate-level power simulation respectively. The power traces are combined with the selected leakage model to compute the leakage time interval, and the leakage estimate for leakage model. Finally, the toggle traces, the leakage time interval, and the leakage estimate are combined into the LIF per cell.

5.3.1 Power Simulation

In this preparation stage, PACA takes RTL design files and generate design netlist through logic synthesis. PACA performs gate-level simulations with a user-defined stimuli. The purpose of this stage is to generate toggle traces (Value Change Dump (VCD)), and subsequently, simulated power traces. PACA uses gate-level power modeling on post-synthesis

or post-layout netlists. Power modeling at the gate-level abstraction level strikes a balance between simulation efficiency and accuracy. It is applicable to the complete chip, while still correctly characterizing sub-cycle-level power effects. In contrast, RTL power modeling or toggle-counting misses many of the important electrical effects in side-channel leakage, and transistor-level power modeling is too complex to achieve at chip-level over extended periods of time. Section 5.10 further elaborates on the simulation accuracy. The experimental results shown in this chapter are made for a 180nm CMOS standard cell technology.

5.3.2 Selecting the Leakage Time Interval

The next step of PACA is to narrow down the time window over which the LIF are computed. The rationale is that we want to determine the LIF over an interval during which the leakage model $L(V)$ is valid and during which side-channel leakage may occur.

The leakage model, in the context of power-based side-channel analysis, is an estimate for the information leakage incurred through power consumption variations. The leakage model L is a function computed over a secret intermediate variable V. The objective of side-channel analysis is to reveal the value of V through many observations of the measured power consumption and correlating those observations with $L(V)$. Popular choices for $L(V)$ are the Hamming Weight or the Hamming Distance on V; the Hamming Weight reflects value-based power leakage in CMOS, while the Hamming Distance reflects distance-based power leakage in CMOS. The objective of PACA is to identify, within a gate-level netlist, those cells that realize $L(V)$. Naturally, there are many possible choices for the leakage function, and PACA makes the assumption that the designer is able to provide $L(V)$. If the algorithm and implementation are known, such a leakage function can always be found. For example, a common choice for $L(V)$ for AES hardware implementations is the Hamming

Distance between the state of different rounds. For AES software implementations, the Hamming Weight of one or a few bytes of the AES state is typically used. However, V does not have to be related to a cryptographic key, and any sensitive value processed in a design could be analyzed. For example, PACA can be used to study bus transfer operations in an SoC. In that case, V is a sensitive value transferred over the bus, and $L(V)$ is the Hamming weight of the value. The Hamming weight reflects the pre-charged nature of a shared bus [120].

We now narrow the search window to the *Leakage Time Interval* using power correlation. We use simulated system-level power traces P and correlate them with the traces from the leakage model $L(V)$. We then compute the correlation ρ as

$$\rho_{L(V),t} = \frac{cov(L(V), P(t))}{\sigma_{L(V)}\sigma_P} \tag{5.1}$$

where:

cov = the covariance

$\sigma_{L(V)}$ = the standard deviation of $L(V)$

σ_P = the standard deviation of P

The Leakage Time Interval is defined as the time window(s) for which

$$\rho_{L(V),t} > \rho_{threshold} \tag{5.2}$$

The threshold level $\rho_{threshold}$ is based on the designer's definition of a distinguishable correlation peak. We can use the Pearson Correlation Confidence Interval to define bounds for $\rho_{threshold}$. Table 5.1 illustrates several choices for $\rho_{threshold}$. Under the hypothesis that the true ρ is zero, the table shows confidence intervals in function of the number of traces (n)

Table 5.1: Pearson Correlation Threshold Levels as a Function Confidence

Confidence Interval	**n=600**	**n=1000**	**n=2000**
99%	±0.105	±0.081	±0.058
95%	±0.080	±0.062	±0.044
90%	±0.067	±0.052	±0.037

and the confidence level. Hence, if the observed ρ falls outside of the confidence interval then we reject the hypothesis and conclude that the design shows leakage.

Because we are computing ρ in a noiseless, controlled environment with full knowledge of the secure asset, we can find sharp correlation peaks with a limited number of traces.

5.3.3 Architecture Correlation for Computing Leakage Impact Factor

Within the Leakage Time Interval, we next perform the architecture correlation as follows. First, we obtain a toggle trace from a gate-level simulation of the design. A toggle trace K_i records the activity of each net i (driven by cell i) using the discrete values -1 and $+1$. If a cell has multiple outputs, then we compute the architecture correlation and leakage impact factor for each output separately. For each time stamp t in the simulation, a toggle trace for net i has the value -1 if the net does not change value, and it has the value $+1$ if the net does change value. We also obtain a toggle trace H that represents the toggle activities of the leakage model $L(V)$.

Architecture Correlation: Next, we perform Architecture Correlation. For each net (or gate driver), we compute the dot product of the toggle trace of the leakage model H with the toggle trace of net i.

Table 5.2: Example of Architecture Correlation

Stimuli	**S0**	**S1**	**S2**	**S3**	C_{ij}
Leakage Model Toggle Activity (H_j)	1	-1	-1	1	
net0 (K_0)	1	-1	-1	1	4
net1 (K_1)	1	1	1	1	0
net2 (K_2)	-1	1	-1	-1	-2

$$C_i = K_i \cdot H \tag{5.3}$$

A high value in C_i has a different meaning compared to a high value in ρ. A high value in ρ reflects a strong dependency between the overall power dissipation and the leakage model. Therefore, a high ρ indicates side-channel leakage. On the other hand, a high value in C_i reflects a strong dependency between activity of net i and the leakage model. A high C_i therefore means that the assumed leakage model is realized by net i. Table 5.2 describes an example computation for the architecture correlation factor C_i. The second row records the toggle activities of the leakage model for different stimuli. The leakage model value toggles for the first stimuli S_0, it does not toggle for stimuli S_1 and S_2, and toggles for S_3. At the same time, net_0 also only toggles on S_0 and S_3 which matches the leakage model in all the four stimuli, therefore, the net_0's correlation score is 4. On the other hand, net_1 and net_2 have a weaker correlations as 0 and -2 respectively. Overall, a more positive and larger architecture correlation indicates that a net approximates the leakage model more closely.

Computing Leakage Impact Factors: The final step of PACA computes the Leakage Impact Factor F_i of the driver of each net i, as the Architecture Correlation of net i, weighted with the average power consumption P_i of the driver of net i normalize by the average power consumption of the whole design P_T , during the leakage time interval averaged over all

stimuli.

$$F_i = C_i \frac{P_i}{P_T} \tag{5.4}$$

This additional weighing factor $\frac{P_i}{P_T}$ is needed because the architecture correlation factor by itself ignores the relative contribution of a cell in the side-channel leakage power footprint. Once the LIF F_i of all cells are determined, they are ranked from highest to lowest. The cells with the highest LIF make the greatest contribution to side-channel leakage. This list can then be used by a designer to efficiently optimize the netlist with countermeasures.

5.4 PACA on encryption subcircuit

This section is an explanatory walk through of PACA operations in detail on a simple design illustrated in Fig. 5.3 including a key-addition and an AES S-box. The design combines an 8-bit secret key k and a 8-bit plaintext p stored in register `key_reg` and register `text_in_reg` respectively. The resulting addition is stored in register `sa_reg` which will drive the input of sbox logic. An additional register is placed in front of the SBOX to separate the sensitive signal `key_reg`⊕`text_in_reg` from other combinational logic. This design uses flip-flops with asynchronous reset, and the testbench asserts reset before every new plaintext and every new key load. We apply PACA using a leakage power model of the output of the key addition, which is expressed as the hamming weight of the key addition result, or $hw(p \oplus k)$. We expect PACA to identify the register `sa_reg` as the major contributor of side-channel leakage, i.e. the cells whose power consumption most closely match the power model. The PACA procedure starts with the collection of power traces of a gate-level model of the design. We collected power traces for 600 random inputs under a fixed key. The power traces are used in a bitwise correlation analysis that matches the leakage model $hw(p \oplus k)$ to the measurements. Fig. 5.4 shows resulting bitwise correlation peak on bit-7 (Most Significant

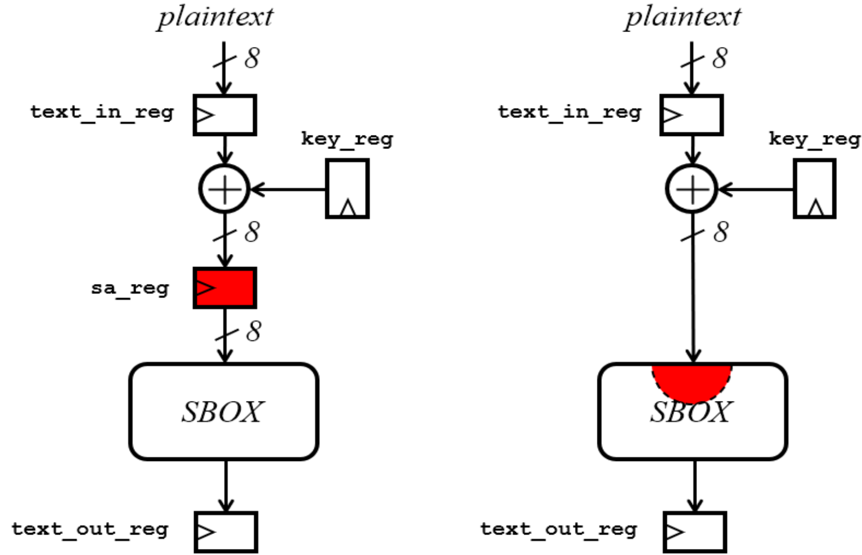

Figure 5.3: (a) AES sbox setup with Register Stages. (b) AES sbox setup without Registers.

Bit). Peak correlation occurs right after `sa_reg` is updated. Using the power traces, we then apply the PACA methodology. PACA computes the Leakage Impact factor for each cell in the overall design. Table 5.3 shows the distribution of resultant LIF for the cells in the whole design (in total 406 cells). The distribution is highly skewed, indicating that only a very small portion of the cells that actually contribute to the side-channel leakage. Among all the cells in the design, `sa_reg[7]` ranks the top in LIF ranking which means PACA identifies the register cells belonging to `sa_reg[7]` as the most leaky cell. This is an expected result, since gates beyond the fan-out of `sa_reg[7]` become less correlated to the power model, and hence contribute less to the side-channel correlation peak.

PACA can further be demonstrated by removing `sa_reg` and running the simulation again.Table 5.4 shows the distribution LIF of the design without stage registers. In this case, the most leaky cells identified by PACA are in the first level of logic of the SBOX. This illustrates that PACA will identify both sequential as well as combinational cells as side-channel leakage sources.

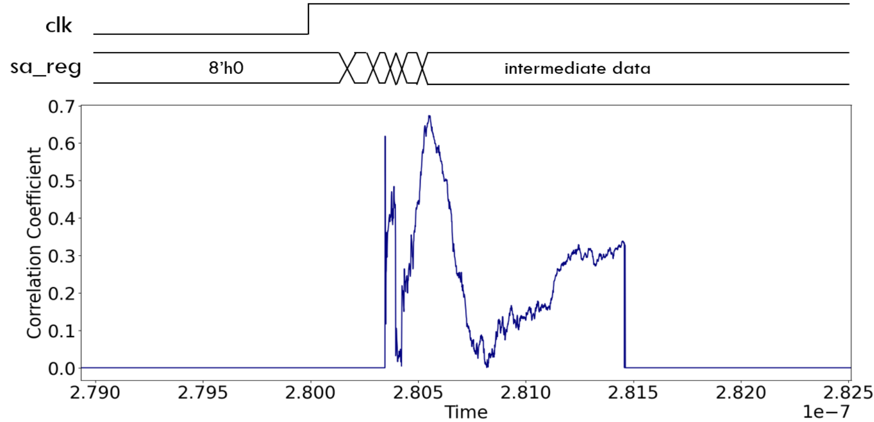

Figure 5.4: Leakage peak for AES sbox with register stage setup. intermediate data = key_reg ⊕ text_in_reg

Table 5.3: LIF Distribution Data for AES sbox with register stage setup

LIF Range	**No. of Cells**
2.3 ∼ 3.0	1
1.6 ∼ 2.3	1
0.9 ∼ 1.6	4
0.2 ∼ 0.9	3
-0.5 ∼ 0.2	397

5.5 PACA on an AES hardware engine

After introducing the insight of PACA, we now apply PACA on an AES coprocessor in this section. The AES implementation runs at one round per clock cycle. The leakage power model used by PACA is the Hamming distance on the previous and current values of one bit in the AES state register. We analyze the output of the first round to find the leakage time interval. Fig. 5.5 reveals a sharp correlation peak when the SBOX output is computed, and we use these correlation peaks to determine $\rho_{threshold}$ at 99% confidence level with 600 power traces. This gives a leakage time interval of 24.6ns (for an AES running at 41.67ns clock period). Next, we perform architecture correlation. Since there are 128 bits of state, there are 128 different leakage models to consider using architecture correlation. In the following,

Table 5.4: LIF Distribution Data for AES sbox without registers stage setup

LIF Range	**No. of Cells**
2.3 $\sim$ 3.0	2
1.6 $\sim$ 2.3	4
0.9 $\sim$ 1.6	8
0.2 $\sim$ 0.9	10
-0.5 $\sim$ 0.2	374

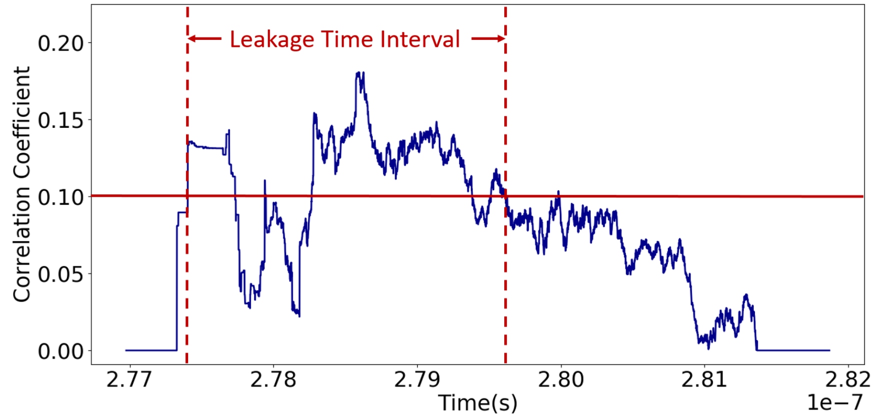

Figure 5.5: Leakage Time Interval for the AES hardware engine.
Leakage Model: HD(AES state bit).

we present the results for a single leaking bit. Our conclusions remain valid for the entire AES state by repeating PACA for each state bit. PACA yields a list of cells in the descending order of their Leakage Impact Factor (LIF) value, which signifies the individual contribution of these cells to side channel leakage.

Table 5.5: LIF Distribution Data for the AES Hardware Engine using HD (AES state bit) as the leakage model

LIF Range	**No. of Cells**
1.9 $\sim$ 2.5	1
1.3 $\sim$ 1.9	1
0.7 $\sim$ 1.3	0
0.1 $\sim$ 0.7	58
-0.5 $\sim$ 0.1	9525

Result Analysis: We analyzed on the cell ranking list from PACA output, Fig. 5.6

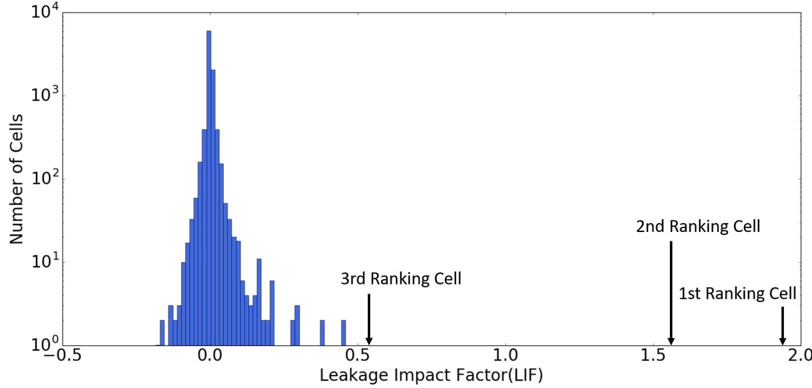

Figure 5.6: LIF Distribution for the AES hardware engine.
Leakage Model: HD(AES state bit); Logarithmic Y scale.

illustrates the LIF distribution for all the cells in the AES design based on the PACA output and Table 5.5 lists the corresponding data. The distribution is highly skewed with only a small amount of cells have high LIF. This indicates that only a small number of cells actively contribute to the side-channel leakage produced following the selected leakage model. The most leaky cell, as identified by the LIF ranking, is a flip-flop of the state-register. Furthermore, the cell ranked just below this register is a cell in the SBOX that is directly driven by this register.

Runtime Evaluation: Table 5.6 shows the runtime overhead of the analysis. We use a 2.3GHz Intel Xeon E5-2699 design server with 128GB of main memory. The complexity of this AES design is 9585 cells. The runtime is broken down into gate-level power simulation (per stimuli), and PACA (per AES state bit). Hence, a full AES design can be analyzed with 600 traces in about 2 hours.

Table 5.6: Runtime Evaluation for AES Hardware Engine (9,585 cells)

Procedure	**Runtime** s/stimuli
Power Simulation	12.28
Architecture Correlation Analysis (per AES bit)	0.268

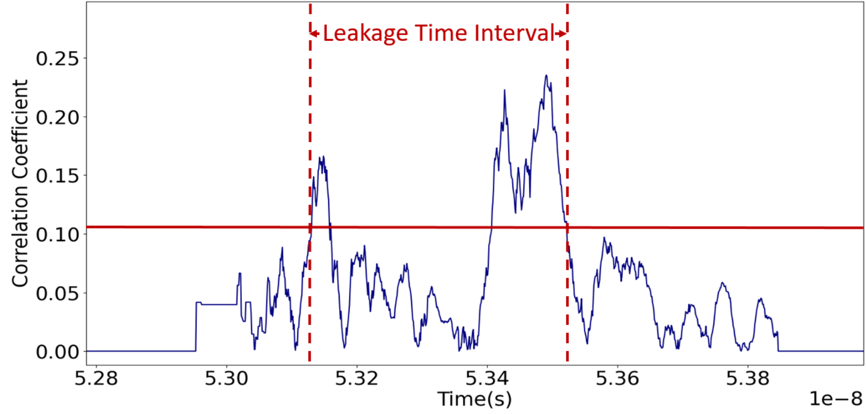

Figure 5.7: Leakage Time Interval for the PRESENT hardware engine. Leakage Model: HD(PRESENT state bit).

5.6 PACA on PRESENT Hardware Engine

We now apply PACA to PRESENT, a light-weight block cipher proposed by Bogdanov et al [24]. Our PRESENT implementation has a 64-bit input, 80-bit key and runs at one round per cycle with clock frequency at 100Mhz. PRESENT has 31 rounds in total for encryption. In this case study, the target leakage model PACA analysis is the Hamming Distance of adjacent round values (second round and third round) in the PRESENT state register. The PRESENT design has in total 653 cells.

After gate-level simulation on PRESENT and implementing correlation analysis on the simulated power traces, we observe a sharp leakage peak (Fig. 5.7). Using a $\rho_{threshold}$ at 99% confidence level for 600 traces (0.105), we find a leakage time interval of 0.41ns (for PRESENT running at 10ns clock period). Next, for this leakage time interval, PACA applies architecture correlation and generates a ranked list cells in the PRESENT design based on the their Leakage Impact Factor value. Without loss of generality to other bits, in the following we present the PACA result for a single leaky bit (bit-7).

Result Analysis: After analyzing the cell ranking LIF distribution for all the cells in the

Table 5.7: LIF Distribution Data for the PRESENT Hardware Engine using HD (PRESENT state bit) as the leakage model

LIF Range	No. of Cells
1.5 ∼ 2	2
1.0 ∼ 1.5	0
0.5 ∼ 1.0	0
0.0 ∼ 0.5	579
-0.5 ∼ 0.0	72

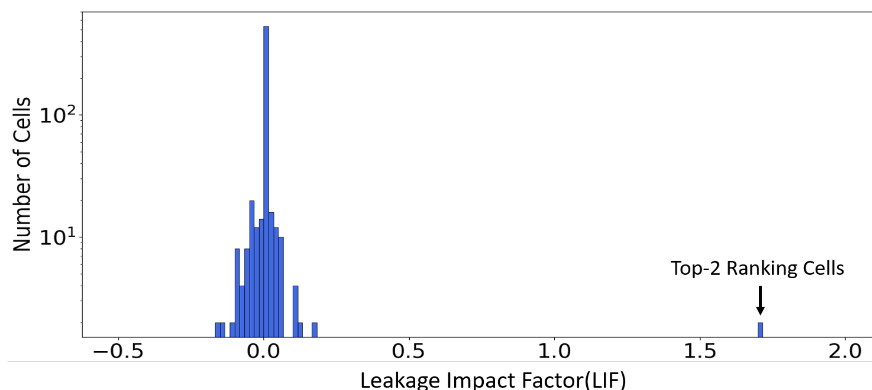

Figure 5.8: LIF distribution for the PRESENT Hardware Engine.
Leakage Model: HD(PRESENT state bit); Logarithmic Y scale.

PRESENT design, we get the LIF distribution as plot in Fig. 5.8 and its corresponding data in Table 5.7. The highly skewed LIF distribution shows that 2 cells stand out from the 653 cells. The most leaky cell is a flip-flop in the state-register and followed by the XOR gate connected to the output of the state-register. After these cells, there is a sharp drop-off in LIF factors, indicating that the remaining cells only contribute marginally to the leakage.

Runtime Evaluation: Table 5.10 shows the runtime overhead of this analysis. The complexity of the PRESENT cipher is 653 cells, a full design can be analyzed with 600 traces in about 1.5 hours.

Table 5.8: Runtime Evaluation for PRESENT Hardware Engine (653 cells)

Procedure	**Runtime** s/stimuli
Power Simulation	4.26
Architecture Correlation Analysis (per PRESENT bit)	0.06

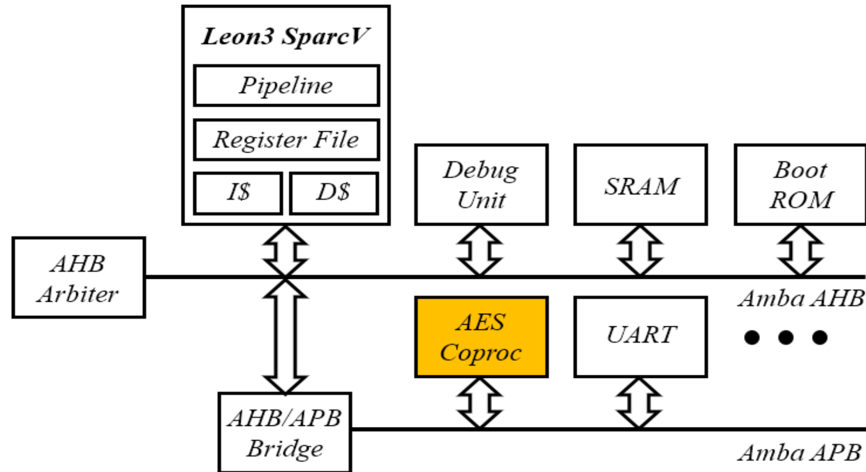

Figure 5.9: SoC block diagram.

5.7 PACA of an SoC Bus Transfer

PACA applies to any activity with a power leakage model. We demonstrate how to analyze the bus interface logic of an SoC for side-channel leakage with PACA. As shown in the Fig. 5.9, the SoC includes a two-level AMBA bus with on-chip memory and several coprocessors, including an AES encryption engine. To perform a hardware-accelerated encryption, the LEON3 writes secure assets (128 bits of plaintext and 128 bits of key material) to the AES coprocessor, triggers the encryption, and waits for a completion flag. The LEON3 then retrieves the ciphertext. A bus transfer affects a large number of components in the SoC, including the caches, the write buffers, the AMBA AHB and APB bus bridges, and finally the memory-mapped interface in the coprocessor. Any of these can potentially contribute to side-channel leakage, and PACA helps to identify which components leak most. We use

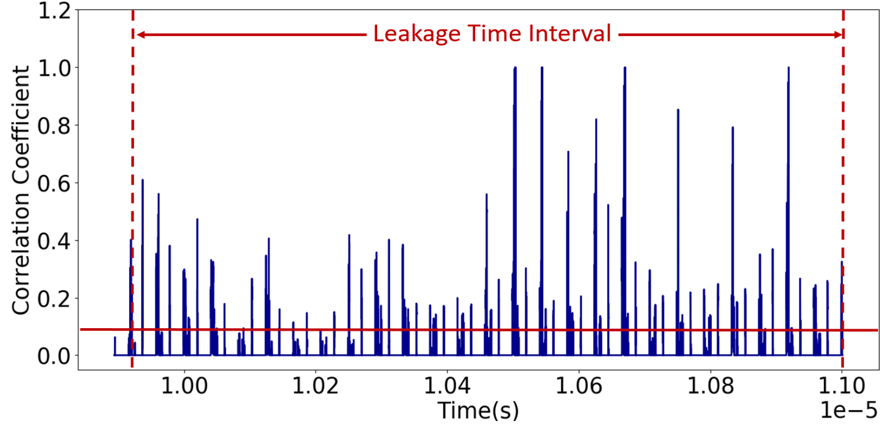

Figure 5.10: Leakage Time Interval for the SoC bus transfer.
Leakage Model: HW(transferred bit).

the Hamming weight of plaintext inputs for encryption as the leakage model. The input data (secure asset) is 128-bit wide, and therefore there are 128 different leakage models to consider. The transfer to the AES coprocessor consists of four 32-bit transfers. Using correlation analysis of the leakage model with the simulated power trace over an interval of these four transfers, we obtain several sharp correlation peaks shown in Fig. 5.10. We use these peaks to fix $\rho_{threshold}$ at 99.0% confidence level for 600 power traces. The leakage time interval is 1.082μs, roughly 26 simulated clock cycles. As before, we present the analysis for a single bit. Since the leakage time interval at the level of SoC covers many different components, we limit the discussion to cells included within the LEON3 core.

Table 5.9: LIF Distribution Data for the SoC Bus Transfer
Leakage Model: HW(transferred bit)

LIF Range	**No. of Cells**
1.9 $\sim$ 2.5	1
1.3 $\sim$ 1.9	0
0.7 $\sim$ 1.3	8
0.1 $\sim$ 0.7	332
-0.5 $\sim$ 0.1	99563

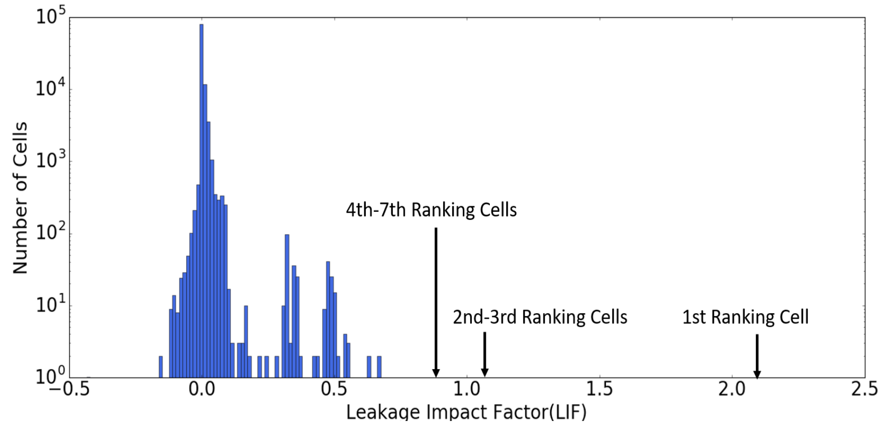

Figure 5.11: LIF distribution for the SoC bus transfer.
Leakage Model: HW(transferred bit); Logarithmic Y scale.

Result Analysis: From the PACA output, we obtained the cell ranking based on LIF. Fig. 5.11 illustrates the LIF distribution for all cells in the SoC and Table 5.9 shows corresponding distribution data. Investigating the results of PACA reveals both expected and unexpected sources of leakage. Top-LIF cells include the flip-flops from the register file, flip-flops from the pipeline operand register of the execution stage, and flip-flops from the pipeline result register of the memory access stage. We notice that cells in the data cache of LEON3 are pointed out by PACA as sources of side channel leakage. This is unexpected because the data cache is disabled by our testbench during the experiment. With the cache disabled, stores of the secure data asset should be directly passed to the memory controller. However, PACA reveals cell activity in the data cache correlating with the secure data asset. Investigation of the specific cells reveals that the leakage is due to a Write Buffer which is integrated in the data cache. The Write Buffer remains active even if the data cache is disabled and is used by LEON3 to ensure that stores do not impede the progress of the execution pipeline by putting pending stores in the Write Buffer. We concluded that identifying such cells would be extremely hard without the systematic analysis offered by PACA. The cells inside the Instruction Trace Buffer (ITB), integrated in the LEON3 core, are another unanticipated source of leakage exposed by PACA on this time window. In our case, LEON3 contains 1

KiloByte of memory as ITB for storing executed instructions. The ITB is implemented as a circular buffer and can hold upto 64 executed instructions. The source of side channel leakage revealed here are the memory cells in the ITB. The ITB is a source of side-channel leakage due to our test mechanism where the plaintext data is a part of the operands in a few of the instructions. These retired instructions end up in the ITB after execution. The existence of the ITB further means that the instructions carrying the secure data asset can persist in the LEON3 core for much longer than intended.

Runtime Evaluation: Table 5.10 shows the runtime overhead of this analysis. The complexity of the SoC is 99,904 cells, 10 times the size of the AES hardware engine. Thus, a full design can be analyzed with 600 traces in about 60 hours.

Table 5.10: Runtime Evaluation for SoC Bus Transfer (99,904 cells)

Procedure	**Runtime** s/stimuli
Power Simulation	329.00
Architecture Correlation Analysis (per AES bit)	32.27

5.8 Selective Countermeasure with WDDL

In the previous section, we demonstrate that PACA can effectively point out the leaky cells from a complex design. And we find that those leaky cells are only a very small portion of cells in the design but actually significantly contribute to the side-channel leakage. In this section, we demonstrate how PACA can be used to implement cost-effective countermeasure by only replacing most leaky cells with its protected version.

5.8.1 Background in Circuit-level Countermeasures

Existing countermeasures against power-based side-channel attacks eliminate or reduce the dependencies between the power consumption and secret information. Secure logic styles are among the first countermeasures developed. Secure logic styles are special logic styles that hide the side-channel leakage by dissipating a constant amount of power. They use balancing techniques, and many variants of them have been developed over the years: WDDL [143], DRSL [46], MDPL [122], LMDPL [50]. Secure logic styles are generally too expensive to be applied across an entire circuit which will cost approximately area overhead of more than 3 times [68]. As the second category, masking countermeasures, apply logic transformations to a design to eliminate the statistical relation between side-channel leakage and power consumption. The masking countermeasure conceals every intermediate value in a circuit as a random number [45, 53, 56, 77]. Such countermeasures remain vulnerable to glitches and cross-coupling [51, 100]. Threshold implementations extend the idea of masking while paying attention to glitches [114]. However, a generic architecture transformation technique that is low-cost and that deals with non-linear circuit effects remains elusive. Threshold implementation requires extra randomness which will cause other issues regarding how much randomness is needed, how frequent the random number needs to be refreshed, etc. The current approach for the aforementioned countermeasure techniques is to apply protection to the entire circuit.

We propose a selective replacement which applies the countermeasure locally to the *individual* leaky cells identified in the previous stage. We first replace these high-LIF cells with equivalent cells that are protected using a hiding countermeasure. The protected cells are based on Wave Dynamic Differential Logic (WDDL), adapted such that a per-cell replacement can be achieved. WDDL is a well-known dual-rail logic style which was proposed as

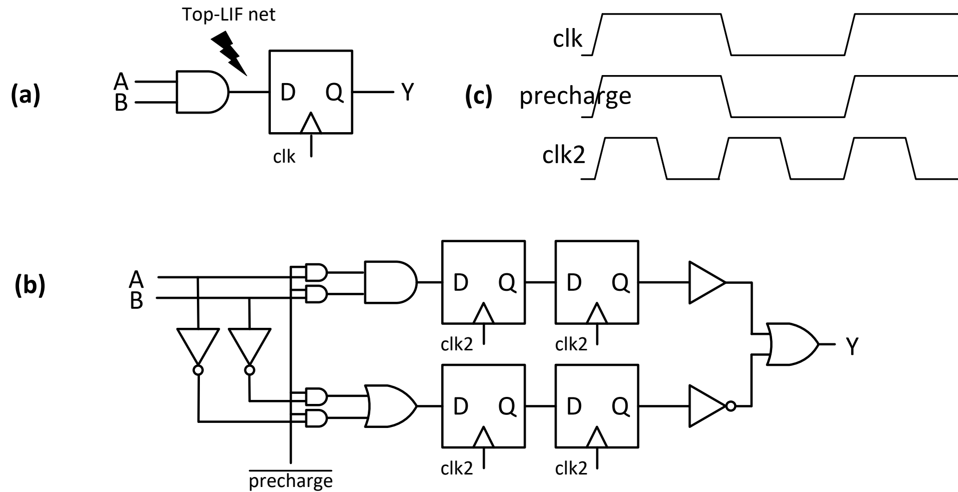

Figure 5.12: Selective-replacement WDDL (a) Original Circuit (b) Transformed Circuit (c) Clocking.

a circuit-level countermeasure against side-channel leakage [143]. WDDL logic ensures that each cell makes a single $0 \rightarrow 1$ transition per evaluation, regardless of the computed value. WDDL cells require a dynamic clocking style with a pre-charge phase and an evaluation phase. Although the feasibility of WDDL has been demonstrated in ASIC, it is expensive. In comparison to unprotected single-rail logic, WDDL occupies 3 times more area and consumes 4 times more power. WDDL is therefore a costly technique to apply chip-wide. When we replace only the high-LIF cells with WDDL versions, the impact on area will be much smaller, while still having a significant impact on the side-channel leakage. We will first explain our countermeasure methodology to implement WDDL on a cell-replacement basis. Next, we evaluate the cost and impact of this countermeasure on the side-channel leakage of the AES hardware.

5.8.2 Selective-replacement WDDL

The WDDL version of a logic cell is created by adding a complementary version of that cell. For example, the AND gate becomes an AND-OR tuple, and a single-rail circuit becomes a

dual-rail circuit with complementary outputs. At the start of every WDDL-evaluation, both rails are precharged to logic-0. Then, the WDDL cell evaluates and a single net in every rail pair switches $0 \rightarrow 1$. To integrate a WDDL cell or a cluster of connected WDDL cells in a single-ended netlist, we add single-to-dual and dual-to-single conversions at the inputs and outputs, respectively, of the protected WDDL region. Every internal net in the WDDL region is protected. Figure 5.12a shows a two-gate circuit with one internal net. Figure 5.12b is the protected version of the same two-gate circuit. As shown in the Figure, the conversion of a single-rail flip-flop to WDDL requires special attention since a flip-flop does not support precharge. We use a master-slave dynamic differential logic [143], which stores the precharge value in a redundant layer of flip-flops. To insert the precharge value, we convert a flip-flop together with its (data-input) driving cell into WDDL. Figure 5.12c illustrates the timing signals of the original circuit and the transformed circuit. A disadvantage of the master-slave method is that it doubles the clock frequency and quadruples every flip-flop. There are many variations and circuit-level improvements of WDDL but these are out of scope for our experiments, which focus on validating PACA.

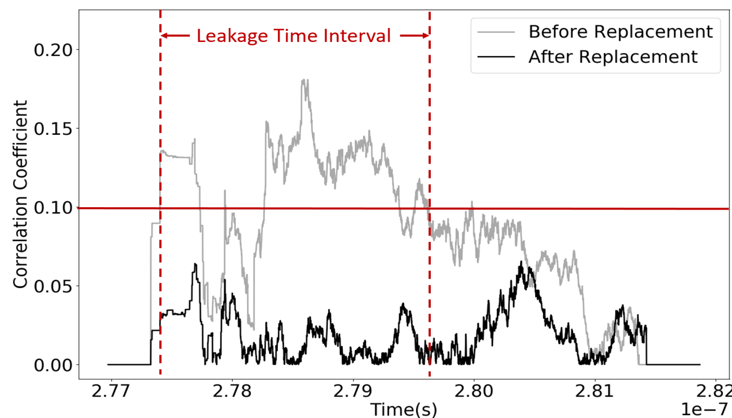

Figure 5.13: Impact on the Pearson Correlation Peak before and after replacing the two top-LIF cells by WDDL

Table 5.11: Impact on the Pearson Correlation Peak under various levels of replacement

Top-LIF cells	ρ_{max}	Cells Added	+Area (+ %)
reference	0.1789	0	0
2	0.0847	282	+8.44
20	0.0586	422	+9.43
40	0.0480	577	+10.54
WDDL[143]	NA	NA	+300

5.8.3 Validation results

Within our AES experiment, we selected top-ranking LIF cells and converted them to WDDL versions while leaving the bulk of the design unprotected. Then, we reran the power simulation and re-evaluated the Pearson correlation under the same power model to detect the impact on the resulting correlation peak. Since the top-ranking cell gate was a flip-flop, we converted the entire state register (128 bits) as well as an output register (128 bits) to a dual master-slave flip-flop, so that we could use a single clock for the entire design. Figure 5.13 shows the effect of replacing just 2 top-ranking LIF cells to WDDL. The correlation is now well below the $\rho_{threshold}$ selected for this confidence level. We also evaluated the effect of replacing additional top-LIF cells. Table 5.11 demonstrates the impact of replacing 2, 20 and 40 top-LIF cell in the design on the peak correlation over the leakage time interval. Although the impact is far less dramatic than the first substitution, a consistent drop can be noticed. The table also indicates the area overhead for this ad-hoc countermeasure, as well as the number of cells we added to the overall design (9,985 cells in total). At only 10% area increase, we are able to obtain a drop of almost four times in the correlation peak. We conclude that PACA helps to identify the cells of a design that cause side-channel leakage.

5.9 Selective Replacement Countermeasure with Decoupling Cell

A disadvantage for the dual-rail approach in selective replacement is that for each cell replacement, single-rail to dual-rail and dual-rail to single-rail interface circuits are needed. The conversion of a single-rail flip-flop to WDDL needs a master-slave dynamic differential logic [143] which doubles the clock frequency. This complicates selective replacement, making a single-cell replacement strategy preferable.

We fuuther investigate another selective replacement countermeasure based on Gornik *et. al.* It is a novel gate-level countermeasure which isolates the power consumption of secure sensitive circuit from main power supply. The isolation is achieved by using a decoupling cell composed of buffering capacitance [68]. The decoupling cell is placed between the main power supply and individual cell. The main advantage of this countermeasure is that it won't cause any performance overhead. However, according to Gornik's strategy, the decoupling cell design has to be applied globally to the entire circuit. By identifying individual leaky cells, PACA can further optimize this countermeasure with applying the countermeasure locally.

We prefer this strategy over an earlier proposed strategy using a dual-rail logic style based countermeasure.

5.9.1 Implementation of the Decoupling Cell

In this subsection, we explain the design of the decoupling cell.

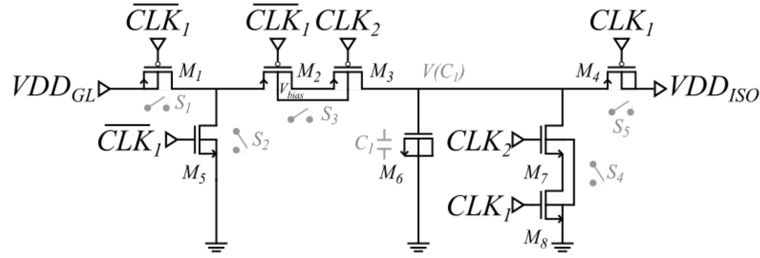

Figure 5.14: Schematic of the decoupling unit.

Topology and Operation

The decoupling cell isolates the power node of the leaky gate from the global supply that powers the rest of the circuitry to mask the leakage. Fig. 5.14 demonstrates the topology of the decoupling unit. This unit is composed of five switches, S1 - S5, and a capacitor, C1. The switches are controlled by clock signals that are adjusted to operate the circuit in three different modes, namely charging (CH), discharging (DS), and buffering (BF) modes. In the CH mode, switches S1 and S3 are closed to charge up the capacitor through the global supply node, VDD_{GL}. The rest of the switches S2, S4, and S5 are open. In the BF mode, the previously charged capacitor powers up the leaky gate. In this mode, switches S1 and S3 are opened, S2 and S5 are closed, and S4 is remained open. Since S1 and S3 are open during the BF mode, the power rail of the leaky gate, VDD_{ISO}, is isolated from the global power rail. Also, S2 is closed to further reduce the power leakage from the leaky gate by shorting the intermediate connection to ground [68]. The capacitor C_1 can supply power to the leaky gate for a certain amount of time as it discharges over time; therefore, it needs to be recharged. However, before recharging and establishing the connection between VDD_{GL}, the capacitor needs to be fully discharged to remove the power data dependency of the leaky gate. The purpose of this action is dissipating the same amount of power in every operation cycle of the decoupling cell to mask the real power consumption of the leaky gate. Hence, after the BF mode, the circuit enters the DS mode to discharge the remaining charge. In this

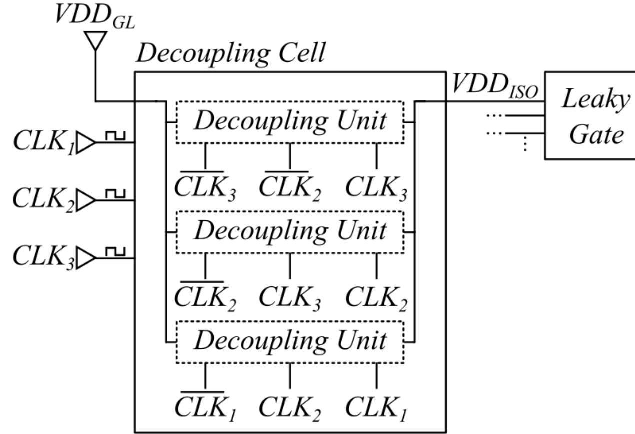

Figure 5.15: Block diagram showing the placement of the decoupling cell.

mode, S4 is closed, and S2 remained closed (again for the enhanced isolation), S5 is opened, S1 and S3 stay open. The aspect ratios of the switch transistors M1-M5 and M7-M8 were adjusted to optimize the speed and power consumption of the circuit. Together with the size of the capacitor M6, the W/L ratio of M4 determines the current capacity and current driving capability of the decoupling circuit, respectively. Thus, we selected the sizes of M4 and M6 to be able to provide the required current to the leaky gate. Increasing the size of the capacitor or the width of M4 may cause an unnecessarily high current driving capability, which increases the power consumption of the circuit. In our design, the power consumption of the decoupling cell is 19.64 nW. We set the aspect ratios of M1-M3, M5, M7, and M8 to 900μm, the highest possible width that can be set in 180 nm technology, to reduce the resistivity of these switches and in turn to increase the speed of charging and discharging modes. Rise and fall times during charging and discharging modes are 253.27 ps and 184.3 ps, respectively. Since the voltage drop on the capacitor ($V(C_1)$ in Fig. 5.14) decreases in the BF mode and the current flow between the decoupling unit and the leaky gate is not continuous due to charging and discharging, we combine three decoupling units to ensure that the leaky gate is supplied with adequate voltage continuously in the entire operation interval. Fig. 5.15 demonstrates the block diagram of the decoupling cell, in which the circuit

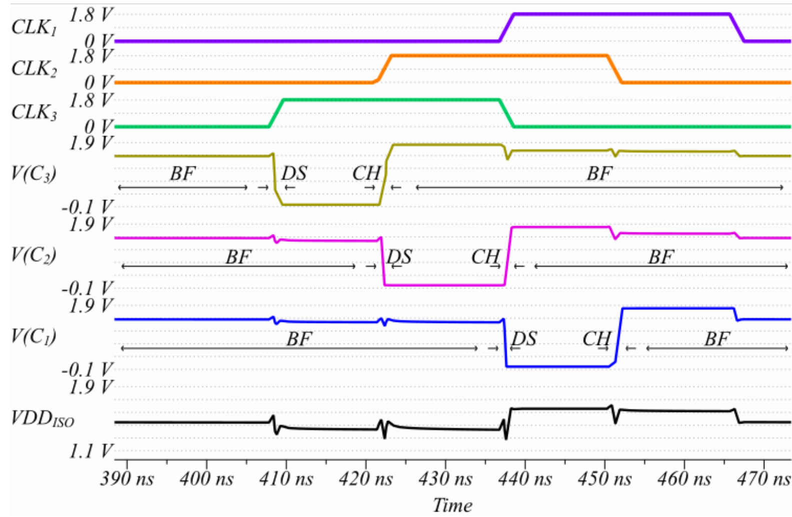

Figure 5.16: Simulation results of the decoupling cell.

structure is the same for each decoupling unit, but the clock inputs change to adjust the timing of the modes (BF, CH, DS) for each unit.

Fig. 5.16 illustrates the clock signals (CLK_1, CLK_2, CLK_3,) for controlling the decoupling cell, voltages on the capacitors ($V(C_1)$, $V(C_2)$, $V(C_3)$,) for each decoupling unit, and the output voltage of the decoupling cell (VDD_{ISO}) connected to the supply node of the leaky gate as shown in Fig.5.15. At least one decoupling unit is in the BF mode throughout the operation as can be seen. Also, one can observe that the output of the decoupling cell varies between 1.25 V to 1.6 V (when the global supply voltage is equal to 1.8 V). The output voltage changes because of the voltage decrease on the capacitors during the BF mode and voltage drops and rises occurring in the DS and CH modes. To minimize the voltage variation, we have modified the timing of the clock signals to have at least two of the capacitor in the BF mode as demonstrated in Fig. 5.16. This will ensure a certain minimum output voltage (1.25 V in our case) and avoid significant voltage drops at the output that may cause to the dysfunction of the leaky gate.

Setup for the Transistor Level Simulations

The power consumption of the s-box, with and without decoupling cell replacement, was analyzed in Cadence Virtuoso Design Environment [32]. The gate-level netlist of the Sbox circuit was imported as a schematic cell view by using the built-in import tool of Virtuoso. A functional cell view written in Verilog Hardware Description Language (HDL) generates the digital inputs of the s-box which are namely the 2-byte plain-text and 2-byte cipher-text, and the clock signal. We used Spectre Analog Mixed-Signal (AMS) Designer [31], to be able to run the Verilog code which is a digital design component in terms of the signals that are produced and observe the power trace of the s-box, an analog signal, in a single simulation environment.

We run simulations both with and without the replacement of the decoupling cell. For the replacement, the decoupling cell was connected to the supply node of the leaky gate in the schematic. For both cases, the current drawn from the global supply node was measured. The results of these measurements essentially gave the power traces, which were then exported from Virtuoso for post-processing and correlation analysis.

5.9.2 Selective Replacement Result

We demonstrate the effectiveness of our proposed selective replacement with our AES sbox experiment. We selected top-ranking LIF cell `sa_reg[7]` identified by PACA and decoupled it while leaving the rest of the design unmodified. Then, we reran the power simulation and re-evaluated the Pearson correlation under the same power model to detect the impact on the resulting correlation peak. Fig.5.17 shows the effect of replacing a single top-ranking LIF cell in the SBOX. The correlation drops dramatically and is now well below the $\rho_{threshold}$

selected for this confidence level.

In terms of the overhead of PACA selective replacement countermeasure, we introduce a single extra decoupling cell in the design but achieve significant improvement in the side-channel security. In the originally proposed decoupling cell methodology [68], the designer needs to decouple every cells in the design. This leads to an increase in the design area by a factor of 10. As we demonstrated in the previous section, only a very small portion of cells in the design actually contributes to the side-channel leakage. Therefore, selective replacement is a highly-targeted and low-cost countermeasure. Traditional countermeasure such as threshold implementation [114], wave dynamic differential logic (WDDL) [143], improved masked dual-rail precharge logic (iMDPL) [123], et al. will at least double or triple the design area. Finally, in terms of performance our proposed approach does not affect the performance, in contrast to traditional countermeasures.

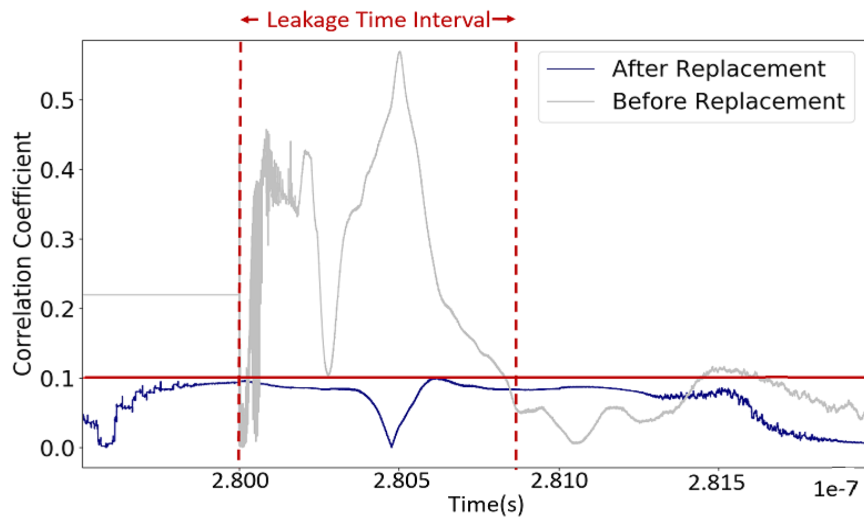

Figure 5.17: Impact on Pearson Correlation Peak before and after replacing only the Top-1 LIF cell by decoupling cell.

5.10 Discussion

In the final section, we elaborate on several concerns relevant to PACA including power correlation vs TVLA, and the comparison of leakage detection by power simulation vs leakage detection by ASIC measurement.

Selection of Leakage Model The problem addressed by PACA is the following. Given an architecture and a *known* leakage model, identify what part of the architecture contributes to the leakage. PACA is an architecture analysis tool instead of a side-channel leakage simulation tool. The PACA methodology is generic and applies to any leakage model. By targeting different leakage models, PACA will reveal the leakage sources corresponding to the selected leakage model. It is up to the designer to select the right leakage model to identify leaky gates. However, these models are commonly known. Internal and external security testing labs estimate the strength of an implementation using state of the art side-channel attacks either on silicon or through simulations. Such attacks typically use leakage models and therefore the designers can obtain the knowledge of the 'right' leakage model as a result of the testing effort. Applications such as AES have well-known leakage models. For example, the Hamming Distance of the adjacent rounds outputs of AES is a typical leakage model, which can be used by PACA. In our setup, we iterate through all leakage models (all 65 combinations of input data and intermediate values) of the AES application and we choose the leakage model which gives us significant correlation peaks which can then be used for analysis using PACA. To analyze the bus transfer procedure of a microprocessor, the Hamming weight model is chosen because during bus transfer the power consumption depends on the Hamming weight of the secret data [120]. PACA is developed to solve known security vulnerabilities. Exploring unknown vulnerable leakage models for the design is not

in the scope of PACA and is not discussed in this chapter.

We acknowledge that statistical detection methods, such as TVLA, can demonstrate the presence of sensitive variables in a power trace while avoiding the difficulty of choosing a leakage model. However, we opted not to rely on TVLA for PACA for the following reason. TVLA does not establish the likelihood of *exploitable* leakage [149]. There is no obvious relationship between the leakage peaks detected by the TVLA and the existence of an attack. In contrast, power correlation based on a leakage model can always be used as a distinguisher for an attack. A power correlation peak reflects the actual difficulty of key recovery. Furthermore, unlike TVLA, power correlation with a leakage model has a precise interpretation in terms of the gates in a design.

Power Correlation vs TVLA Statistical based side-channel detection method, such as TVLA, can demonstrate the presence of sensitive variables in a power trace. However, TVLA indeed has its own short-commings. The most notorious one being the lack of an obvious relationship between the leakage peaks detected by the TVLA and the exploitability and efficiency of it in attack. Another problem of TVLA is the false negatives/false positives, i.e. TVLA fails to detect the leakage while the leakage exist/detects the leakage while the leakage does not actually exist. Therefore, it's hard to guarantee that the designer are trying to solve the problem that are actually exist with TVLA. Power correlation is always used as a distinguisher for attack. Therefore, power correlation peaks reflects actual difficulty of key recovery. Furthermore, unlike TVLA, power correlation has a precise interpretation in terms of the gates in the netlist of a design. Therefore, we use power correlation rather than TVLA as the side channel leakage evaluation tool.

Power simulation vs ASIC measurements PACA enables the designers, at early design-time before chip tape-out, to the identify side channel leakage source and efficiently

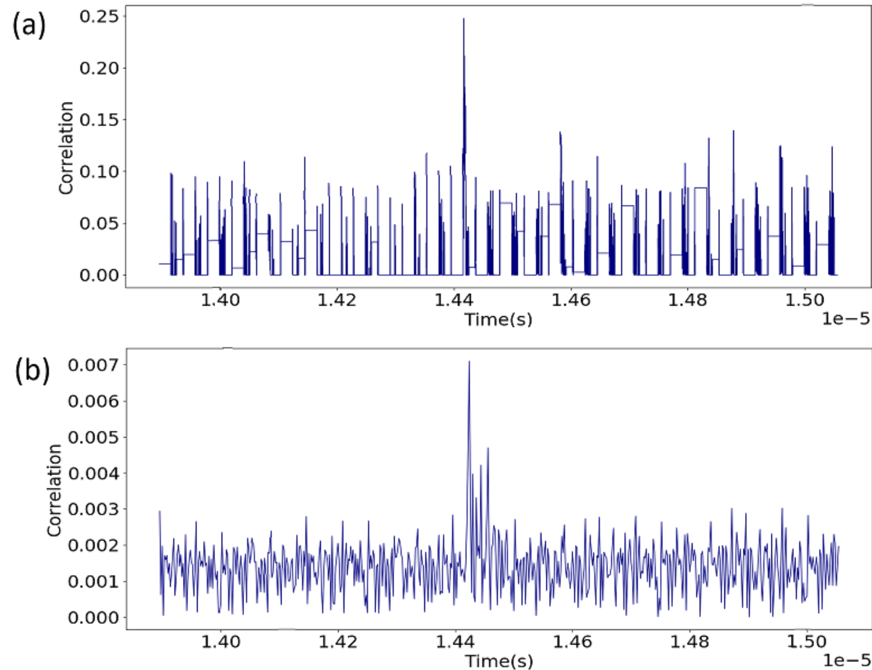

Figure 5.18: Correlation results for the AES Coprocessor using HD(AES state bit) obtained from (a) Simulated Traces, (b) ASIC Measurement Traces.

fix a side-channel leakage vulnerability.

In order to evaluate the *accuracy* of the design-time power estimation, we measure an ASIC prototype of a non-remediated design [158] and we compare this to our simulated traces. We confirm that the correlation peaks identified using PACA correspond to those identified in the ASIC measurement. Furthermore, due to the absence of measurement noise, the correlation peaks from PACA are sharper, and require fewer traces, compared to the correlation peaks from ASIC measurements. The presence of noise in ASIC measurement traces make side-channel leakage assessment difficult, while highlighting the advantages of simulated trace.

PACA allows identifying the side channel leakage source at design-time, and before chip tape-out. PACA also efficiently fixes the identified side-channel leakage vulnerability.

Fig. 5.18 shows the leakage for the AES hardware engine in the first case study. The figure compares the correlation peaks resulting from 500 simulated traces to the correlation peaks

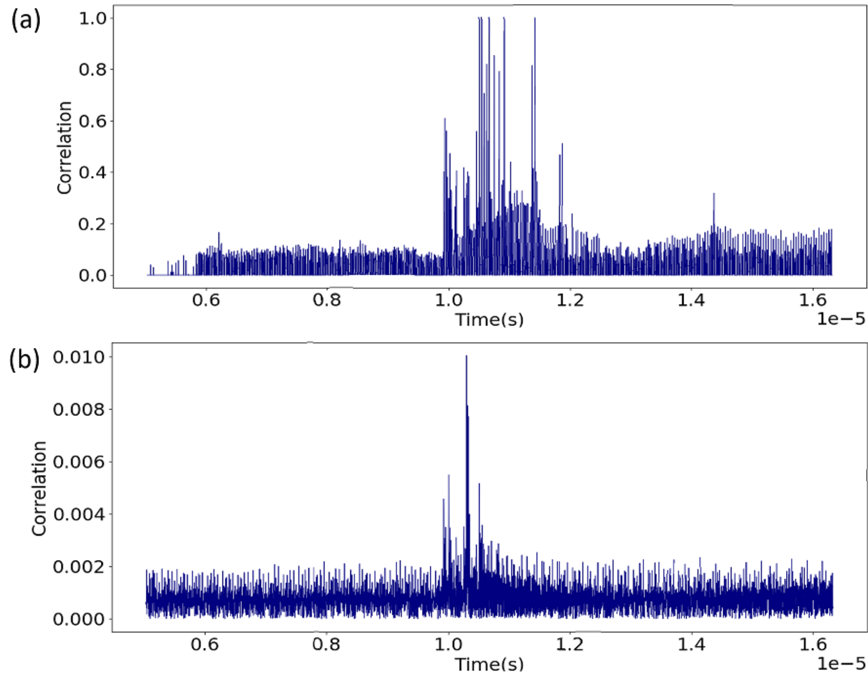

Figure 5.19: Correlation results for the SoC Bus Transfer using HW(transferred bit) obtained from (a) Simulated Traces, (b) ASIC Measurement Traces.

resulting from 500,000 measured traces from an ASIC implementation of the same design. We can observe that both in the ASIC measurement and simulated trace leakage peaks can be detected. The time interval during which correlation peaks appear in the simulated trace is aligned with the time interval in the ASIC prototype measurement.

Fig. 5.19 shows the leakage for the SoC bus transfer leakage model in the second case study. Correlation peaks of power traces with input data can be observed in both the ASIC measurement traces and the simulated traces starting at the same period of time. However, as compared to the simulated traces, the ASIC traces are noisy which leads to fewer and smaller correlation peaks.

These comparisons confirm that PACA's analysis results reflects the leakage from the ASIC measurement. In general, simulation traces are much less noisy compared to the ASIC measurement. Therefore, it requires a fewer number of traces to detect the leakage. Addi-

Table 5.12: Power Simulation Levels Trade-offs

Simulation Level	**Simulation Accuracy**	**Simulation Speed**	**Side-channel Leakage can Capture**
RTL	low	fast	logic transition
Gate	medium	medium	logic transition + glitches + static power
Transistor	high	slow	logic transition + glitches + static power + parasitics

tionally,because of the absence of noise, the simulated traces can detect more leakage peaks compared to actual ASIC measurement. Therefore, simulated traces reflect the worst-case scenario. It will overall help the designer decrease the false-negative cases.

Table 5.12 illustrates side-channel leakage modeling at three different modeling abstraction levels: transistor-level, gate-level, register-transfer level (RTL) [155]. These modeling abstraction levels apply varying degrees of modeling precision to time and data in order to improve the simulation performance. At the most detailed transistor-level, behavior is modeled using continuous-time and using (continuous-value) circuit equations. At higher abstraction levels, behavior becomes increasingly discrete and abstract. Time is abstracted into discrete events (gate-level), clock cycles (RTL) and data is abstracted into bits (gates, RTL). Abstraction of time and data has a significant impact on the accuracy of power modeling, and consequently on the accuracy of side-channel leakage estimation. A broad range of power-related effects have shown to create data-dependent side-channel leakage. This includes dynamic power consumption (net transitions), static power consumption (leakage) [108], glitches [99], and coupling [33]. Table 5.12 observes that not every abstraction level is able to capture every form of power-based side-channel leakage, and that lower, more

detailed abstraction levels become more comprehensive in modeling of power-based side-channel leakage. However, the main source of side-channel leakage comes from logic transitions. Second-order effects in the circuits, such as static power and parasitic effects can also cause side-channel leakage, however, not as significant as logic transitions. Capturing these effects requires a significant increase of the simulation detail.

To identify the source of leakage, PACA operates at the gate-level, which offers a good trade-off between design abstraction (simulation speed) and side-channel leakage modeling detail. It is applicable to the complete chip, while still correctly characterizing sub-cycle-level power effects. In terms of evaluating the effectiveness of our proposed countermeasure through simulation, we adopt transistor-level simulation which is the most accurate simulation level in this work. Some low-level leakage sources, such as cross-talk of the wires, and parasitic coupling, are known to cause masking-based countermeasures [100]. Our proposed countermeasure is a hiding-based solution which is not vulnerable to these low-level circuit effects.

5.11 Conclusion

PACA is a significant step towards secure design automation. The PACA methodology helps not only to identify gate-level side-channel leakage issues in the early stages of an IC design, but also precisely pin-point the leaky cells in a complex design. PACA further helps to mitigate side-channel leakage with low cost by selective replacement of the highest-leaking cells of a design. Through examples at various levels of abstraction, we demonstrated the scalability and feasibility of PACA.

Chapter 6

Augmenting Leakage Detection with Boostrapping

In this chapter, we will present an improvement on the existing side-channel leakage assessment based on a statistical method - Bootstrapping. This work has been published in Constructive Side-Channel Analysis and Secure Design: 11th International Workshop, COSADE 2020 [156].

6.1 Introduction

One of the first side-channel attacks in the literature was described by Kocher [85], who noted that the time required to compute an RSA signature could reveal the private key. Further work [86] demonstrated the feasibility of determining cryptographic keys by analysing the instantaneous power consumption of a cryptographic device, since individual bits of key related intermediates influence the power consumption. This type of attack is typically referred to as Differential Power Analysis (DPA) and has evolved since to include various statistical tools. Despite its name, the technique is not limited to the usage of power measurements but can be applied to other measurements of physical properties of a device that is affected by the handling of secret related data, for example, electromagnetic radiation [61, 127]. Side-channel attacks have since been demonstrated on a wide variety of devices ranging from small

single purpose chips [16, 43, 80] to large general purpose SoCs and CPUs [63, 107, 145].

Evaluating the first countermeasures to DPA typically involved the labor intensive task of performing all known attacks and/or evaluating the power distribution of all sensitive intermediates for data dependencies. However, this does not guarantee there is no side-channel leakage, only that certain attacks are not possible or that certain intermediates are not present. In 2011, Test Vector Leakage Assessment (TVLA) was proposed by Goodwill et al. [67] as a means to detect side-channel vulnerabilities by determining if side-channel leakage can be detected. Goodwill et al. described using Welch's t-test with a null hypothesis that no leakage was present in a pointwise comparison of two sets of power consumption traces. Any t-statistic greater than 4.5σ (corresponding to a false positive rate of 1×10^{-5}) would indicate the presence of leakage and that an attack may be possible. This allowed designers to expose side-channel leakage without conducting potentially complex key recovery attacks. It has become a popular tool for the assessment of side-channel vulnerabilities in secure devices because of its simplicity, and Balash et al. [17] later showed that these methods allow us to determine that microprocessors leak in many, often unexpected, ways.

TVLA is widely used by research and testing facilities even though it does have several shortcomings. One being the lack of an obvious relationship between the leakage and the exploitability and efficiency of using the leakage in an attack. Another one is the existence of false negatives which occur when the measurements contain side-channel information but TVLA fails to detect it. This can happen in a couple of scenarios. First, it is possible that two data sets have very similar means and the amount of measurements is not sufficient to discern a difference. Second, the data sets obtained could be very noisy (low Signal-to-Noise Ratio (SNR) [102, 160]) and again the number of measurements could not be sufficient to discern a difference. Typically, one would try to reduce the risk of a false negative by applying TVLA on a large number of measurements [17, 138] or using multiple input vectors for the

fixed set. One can also opt to use a fixed versus fixed test which will reduce the algorithmic noise but has the added drawback that some leakage may not show up due to the choice of vector [138]. Ideally, the confidence in the outcome of the evaluation can be improved by repeating the TVLA test multiple times over new measurements. However, this increases the total number of measurements and evaluation time. In practice, one typically only has a limited number of measurements, or available time, so it seems opportune to extract as much information as possible from a set of acquired traces. In this work, we seek to provide that, without the need for more measurements. We achieve this by improving the efficiency of the data usage or, looking at it from a different angle, one could argue that we decrease the the number of measurements needed for detecting leakage.

Bootstrapping is a computer-based technique for statistical inference proposed by Efron [42]. It can be used to estimate the statistics of a population by repeatedly re-sampling from the same data set. The repeated sampling provides an approach that enables repeated testing from one set of measurements, where we only require that the measurements are representative of the population from which they are drawn. We demonstrate that this method, applied to side-channel leakage detection, can reduce the number of traces required to detect leakage by one, or more, orders of magnitude, and is demonstrated with multiple experimental setups. First, we generate simulated traces with the knowledge about the side channel leakage and we apply bootstrapping and leakage detection on our simulated traces to gain an understanding of the increase in efficiency. Then, we apply bootstrapping on actual measurements from both software and hardware implementations with and without side-channel countermeasures. Based on our experimental results, we show that bootstrapping can significantly reduce the number of traces required to detect leakage by building a degree of trust in the results from the leakage detection test. Our proposed method provides another metric and gives complementary information about the leakage to the evaluator.

This chapter is organized as follows. Section 6.2 describes the preliminaries needed to follow the story in the chapter. Section 6.3 details on how to apply bootstrapping to leakage detection tests. Results are given in simulation and on a variety of software and hardware implementations. The limitation and interpretation of the bootstrapping is elaborated upon in Section 6.4. Some implementation details are given in Section 6.5.

6.2 Preliminaries

We first provide an introduction to the methods we will use throughout the text.

6.2.1 Leakage Detection using Welch's t-test.

Welch's t-test is a statistical test used to compare sample means of two sets with, possibly, unequal variance but still under the assumption of normality. The output of the test provides a test statistic which can be combined with a threshold to validate the null hypothesis H_0 that both sets have equal means, or state there is no evidence supporting the null hypothesis so the alternative hypothesis H_a holds. We consider sets A, B of size n_A, n_B, with means μ_A, μ_B and standard deviation σ_A, σ_B, respectively. With these notations the null hypothesis and the alternative hypothesis are noted as follows,

$$H_0 : \mu_A = \mu_B \quad H_a : \mu_A \neq \mu_B \tag{6.1}$$

and the t-statistic is calculated with the following formula:

$$\psi = \frac{\mu_A - \mu_B}{\sqrt{\frac{{\sigma_A}^2}{n_A} + \frac{{\sigma_B}^2}{n_B}}} \tag{6.2}$$

where $\psi \sim t(0, \nu)$ with ν degrees of freedom and $t(x, y)$ the t-distribution with mean x and standard deviation y. In practice, we use the result that the t-distribution is asymptotically equivalent to the standard normal distribution as the degrees of freedom increase, i.e. we assume $\psi \sim N(0, 1)$. We then transform the t-statistic into a p-value using the Cumulative Density Function (CDF) to argue about the validity of H_0.

Goodwill et al. [67] proposed to use Welch's t-test to detect leakage in implementations of cryptographic algorithms by comparing two sets of side-channel acquisitions. One set would be acquired with a fixed input and the other with a random input. Welch's t-test can be computed point-wise on the acquisitions. A null hypothesis is formulated at each point individually assuming independence of the points. Intuitively one can see that if the means of those two sets (or the distributions) are not equal, the power consumption is data dependencies and could potentially leak information.

Goodwill et al. [67] proposed a Type I error, a false positive, rate of 1×10^{-5}, meaning the two-tailed p-value $p < 1 \times 10^{-5}$ would stipulate there is no evidence H_0 is true. This corresponds to the absolute value of $|\psi| > 4.5$ indicating that there is no evidence H_0 is true, and the the alternate hypothesis may be true. In practice, Welch's t-test is applied point-wise across a set of acquisitions so the probability of seeing at least one Type I error is significantly larger than 1×10^{-5}. Ding et al. [160] proposed adjusting the threshold by taking the trace length (total number of points in a measurement) into consideration. For ease of expression, we will use the threshold defined by Goodwill et al. [67], but a different threshold may be appropriate when applying this method.

6.2.2 The Bootstrapping Method.

The bootstrapping method is a computation-based statistical tool proposed by Efron [42] to make inferences about a population parameter based on a sample set. It is typically used to estimate statistical distributions and to quantify uncertainty, under the assumption that the sample set is representative of the population.

Given a set of observations S_{obs} consisting of n samples, $\{s_1, \ldots, s_n\}$, from a given population we can apply bootstrapping by repeated sampling, with replacement, from S_{obs}. This process can be repeated b times, producing b sets $\{S'_1, \ldots, S'_b\}$, where b is chosen arbitrarily. More explicitly, we detail this process in Algorithm 5, where we define the operation $\overset{R}{\leftarrow}$ as taking a random sample from a set. Statistical tests can then be applied to each of these sets producing a set of statistics, which can allow a better analysis than just relying on the observed set S_{orig}.

Algorithm 5: Generating Bootstrapping Sets

Input: $S_{obs} = \{s_1, \ldots, s_n\}$ with $n, b \in_{>0}$
Output: $\{S'_1, \ldots, S'_b\}$

```
1 for i = 1 to b do
2     for i = 1 to n do
3         s'_j <-R- {s_1, ..., s_n} ;
4     end
5     S'_i <- {s'_1, ..., s'_n} ;
6 end
7 return {S'_1, ..., S'_b}
```

Pattengale et al. [118] recommend repeating this process 100–500 times to get a robust description of the distribution of the population. In our work we show that far fewer iterations are required for leakage detection.

6.2.3 Kolmogorov-Smirnov Test

In this chapter, we also apply the one-sample Kolmogorov-Smirnov test (KS test), which is a measure of the difference between a sampled distribution and a defined distribution. The null hypothesis of the test H_0 is that the samples come from the defined distribution, with the alternative hypothesis H_a that the samples have a different distribution.

Let $(s_1, s_2, ..., s_n)$ be the samples in a data-set. For any number x, the empirical distribution function value is the fraction of the data that is smaller x:

$$F_n(t) = \frac{1}{n}\sum_{i=1}^{n} I_{\left\{s_j \leq x\right\}} \tag{6.3}$$

Where I is the indicator function. The test statistic D exploits the maximum distance of the empirical distribution from the sampled distribution and the defined distribution:

$$D = \underset{x}{sup}\,|F_n(x) - G(x)| \tag{6.4}$$

Where G computes the CDF of the defined distribution and *sup* is the supremum function. After getting the D statistic for the KS-test, the corresponding p-value can be calculated from the CDF of the one-sample Kolmogorov-Smirnov distribution.

6.3 Applying Bootstrapping to Leakage Detection

In this section, we describe how we apply bootstrapping to leakage detection. Without loss of generality, we discuss our results using Welch's t-test, since the same method could be applied to any other test that produces a p-value. That is, similar improvements would be

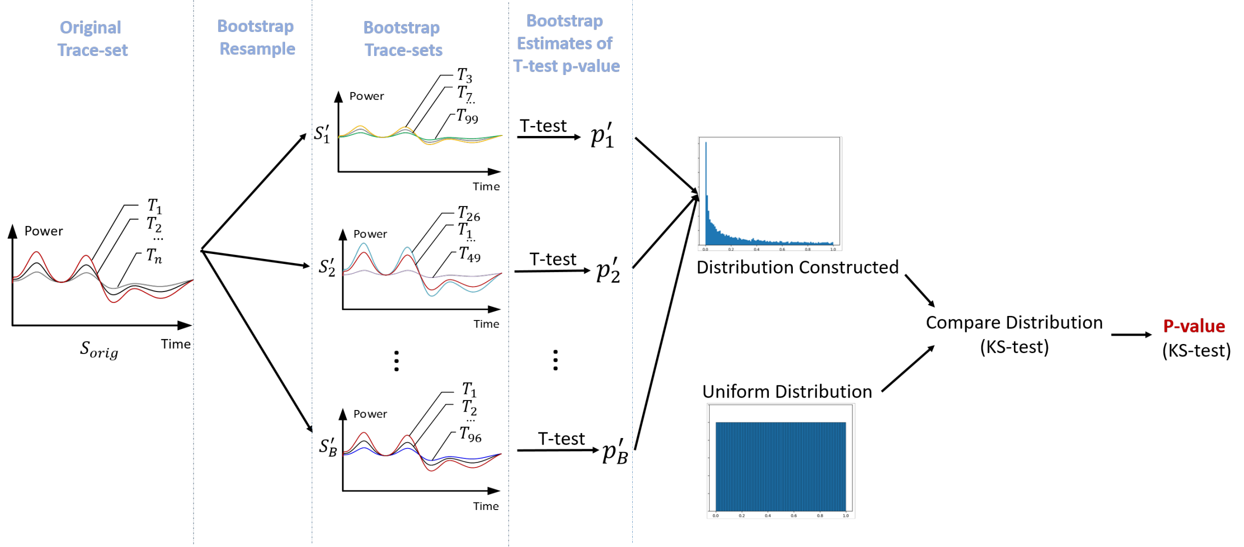

Figure 6.1: Bootstrap Leakage Detection Enhancement

seen if one were to use other statistical tests, such as the χ^2 test [110], Hoteling's T^2-test or Diagonal-test(D-test) [30].

Let $S_{obs} = \{s_1, \ldots, s_n\}$ be the set of n acquisitions to be used in a leakage detection test, as described in Section 6.2.1. Each s_i, for $i \in \{1, \ldots, n\}$, consists of an acquisition and the corresponding metadata indicating whether it belongs to set A or B. We apply bootstrapping, as shown in Algorithm 5, to S_{obs} to provide b sample sets $\{S'_1, \ldots, S'_b\}$, where the choice of b is arbitrary. We then conduct Welch's t-test on each set and compute the resulting p-value, giving $\{p'_1, \ldots, p'_b\}$. Each p-value represents a test with

$$H_0 : \text{no leakage} \quad H_a : \text{leakage} \tag{6.5}$$

and we wish to combine the p-values to test this null hypothesis. Figure 6.1 demonstrates the proposed methodology.

In general, the p-value is a measure of evidence on whether the null hypothesis is true, where a p-value close to 0 can be taken as lack of evidence that the null hypothesis is true, and that

the alternate hypothesis may be true. By definition, if the null hypothesis is true then the p-value is uniformly distributed over the interval $[0, 1]$. It has been shown that the p-value distribution is highly skewed when the alternative hypothesis is true [75].

In this work, we use the distribution of the p-values $\{p'_1, \ldots, p'_b\}$ to evaluate whether there is evidence that the null hypothesis is true. That is, if the null hypothesis is true then

$$\{p'_1, \ldots, p'_b\} \sim U(0, 1)\,.$$

We can test whether this is the case using the one-sample Kolmogorov-Smirnov test to compare $\{p'_1, \ldots, p'_b\}$ to a uniform distribution. In the KS-test we have the null hypothesis that the data-set is drawn from the defined distribution, and the alternate hypothesis that it is not. That is,

$$H_0 : \{p'_1, \ldots, p'_b\} \sim U(0, 1) \quad \text{and} \quad H_a : \{p'_1, \ldots, p'_b\} \nsim U(0, 1)\,. \tag{6.6}$$

The resultant KS test statistic reflects the similarity of the distribution of the p-values with the uniform distribution. As proposed by Goodwill et al. [67], we shall assume the significant level α of 1×10^{-5}, and reject the null hypothesis if the p-value return by the KS-test gives $p < 1 \times 10^{-5}$.

6.3.1 Simulating Leakage Detection

To demonstrate the effectiveness of our method we simulated a single sample, i.e. a simulated acquisition with a trace length of one. We generated sets of data where the sample is the Hamming weight of an 8-bit value with added Gaussian noise to achieve a signal-to-noise ratio of 1 dB, to simulate the setup in the practical environment in which the traces are

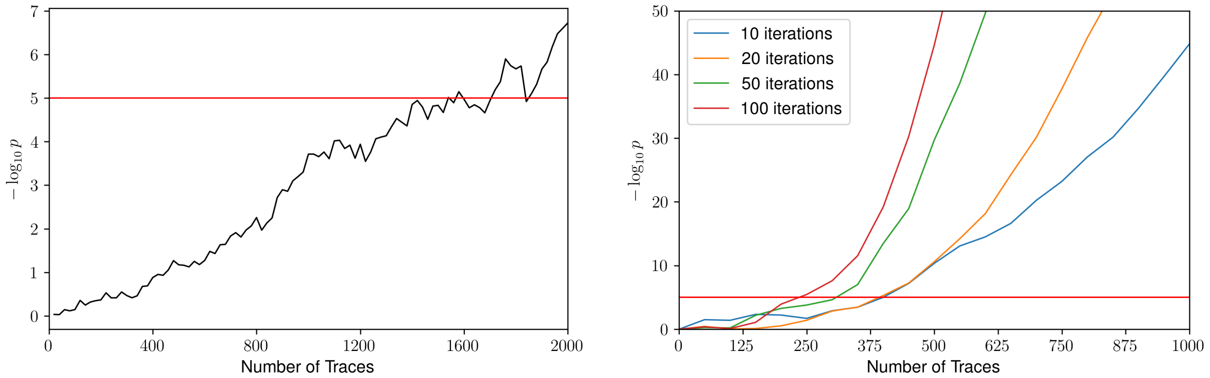

Figure 6.2: The evolution of the p-value with increasing number of traces for TVLA (left) and with bootstrapping (right) using simulated traces

noisy and multiple traces are needed for the t-test to reach the threshold.

In Figure 6.2, we show how the t-statistic produced by TVLA evolves as the number of traces increases, compared to the evolution of the p-values produced by the KS test, as described above. As proposed by Moradi et al. [110], we plot the negative logarithm base 10 of the p-value in both cases. This allows for a simple comparison and the 4.5σ threshold becomes 5. In our simulation, a straightforward implementation of TVLA will show leakage after 1600 traces. If we apply bootstrapping we can see leakage from 200 to 400 traces, depending on the number of iterations of the bootstrapping method that are applied.

To demonstrate why this occurs we generated three sets of single-point traces: Trace-set-A is calculated as the fixed value 5. Trace-set-B and Trace-set-C are calculated from the Hamming weights of 8-bit random values. As above, we added Gaussian noise to achieve a signal-to-noise ratio of 1 dB. In Figure 6.3, we can see two plots of frequency versus p-value, where the p-values are generated from 5000 iterations of the bootstrapping method on 1000 samples. The left plot is the result of applying bootstrapping to TVLA applied to Trace-set-A and Trace-set-B, and the right plot from applying bootstrap enhanced TVLA applied to Trace-set-B and Trace-set-C. These tests represent the fixed-versus-random case and a

comparison case of random-versus-random. In each case the resulting p-values are grouped into bins defined by dividing up the interval $[0, 1]$ into 100 equally sized bins. The difference in the observed distributions is quite striking.

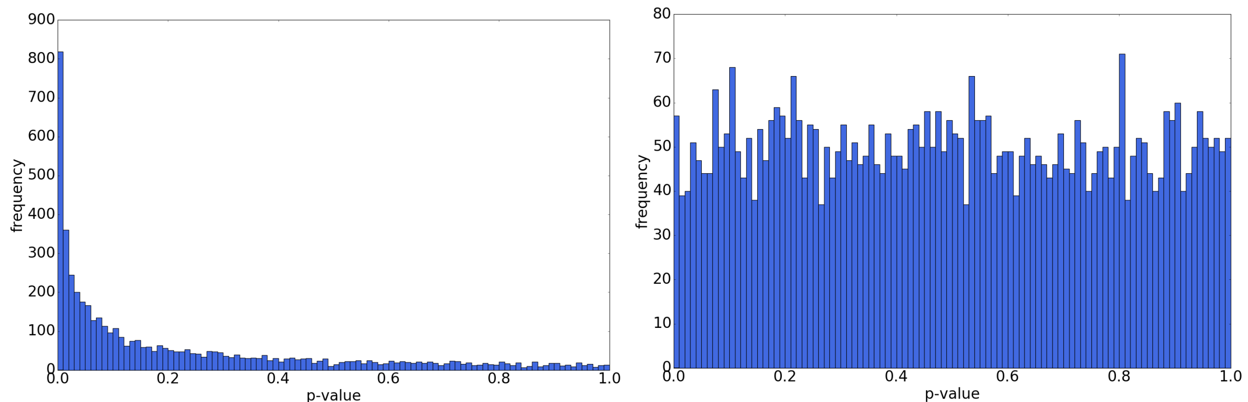

Figure 6.3: The sample distribution of the p-values taken from 5000 iterations of the bootstrapping method applied to samples where a the null hypothesis is false (left) and true (right)

6.3.2 Experimental Results

We then performed experiments to evaluate the practical benefits of bootstrapped enhanced TVLA on a variety of implementations and platforms.

Software AES with Boolean masking. The first experiment is an application of the proposed test to a naïve implementation of a Boolean masked AES on an NXP LPC2124, a 16/32 bit ARM7TDMI-S chip. The implementation was a straightforward 8-bit implementation making use of randomized masked tables for the S-box and the xtime operations. As noted by Balash et al. [17], such implementations are unlikely to be secure. Measurements were acquired with a Langer $RF - U2, 5 - 2$ electromagnetic probe over a decoupling capacitor using a PicoScope 3206D at 400 MS/s with 200 MHz bandwidth.

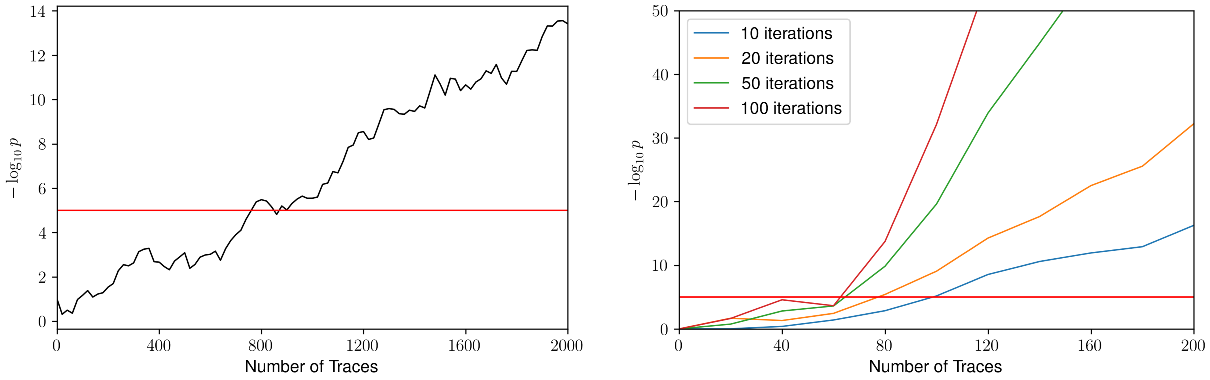

Figure 6.4: The evolution of the p-value with increasing number of traces for TVLA (left) and with bootstrapping (right) applied to an implementation of AES in software

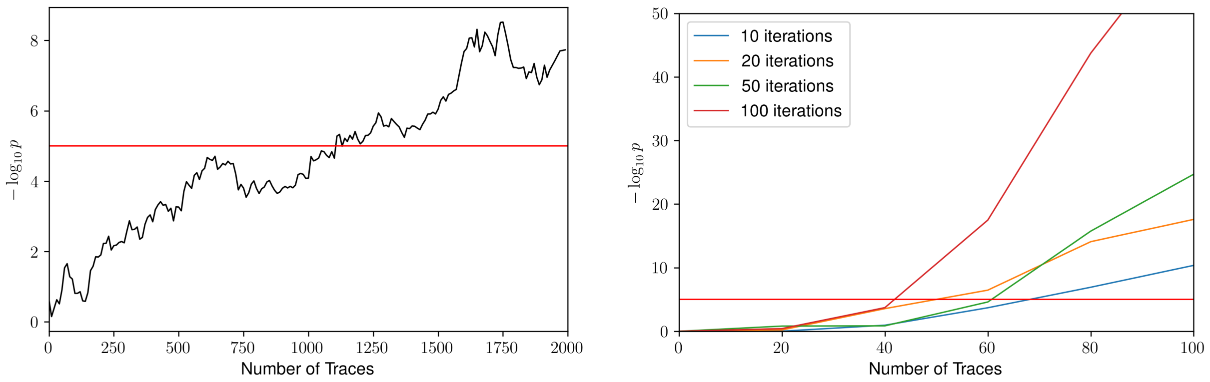

Figure 6.5: The evolution of the p-value with increasing number of traces for TVLA (left) and with bootstrapping (right) applied to an unpotected implementation of AES on an FPGA

Unprotected hardware AES. Our next target was a straightforward single round per clock cycle hardware implementation, i.e. all 16 S-boxes are computed in parallel, on a Xilinx Kintex-7 FPGA. We used a custom FPGA prototyping board where we measured the voltage drop across a measurement resistor using a Tektronix DPO7104C at 1 GS/s.

Lightly protected hardware AES. Our last target was an AES implementation protected with a low-cost dual rail countermeasure implemented on the same FPGA platform as the unprotected AES implementation, described above. AS previously,

we used a custom FPGA prototyping board where we measured the voltage drop across a measurement resistor using a Tektronix DPO7104C at 1 GS/s.

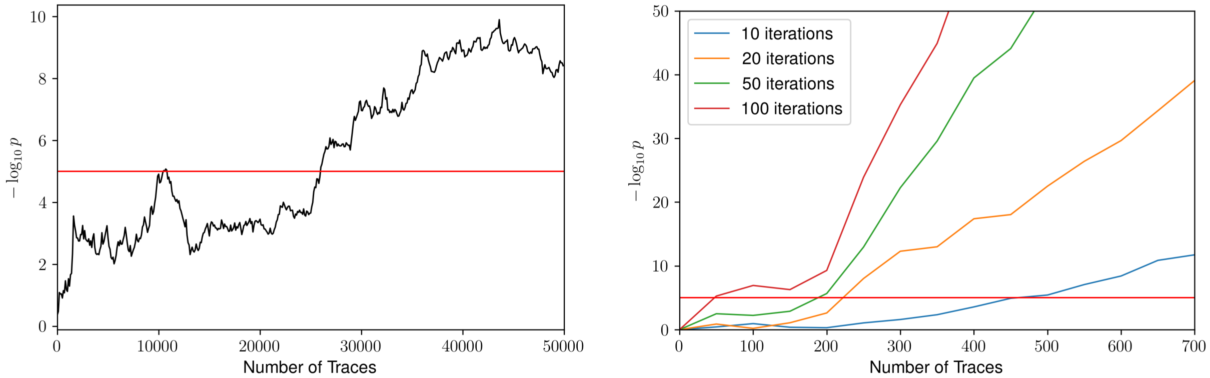

Figure 6.6: The evolution of the p-value with increasing number of traces for TVLA (left) and with bootstrapping (right)

In the three experiments presented above, we can see that the bootstrapping method reduces the number of traces required to detect leakage by at least one order of magnitude in all cases. Or, were we to use all the measurements, we would get with a high certainty all the leaking points this set could uncover. the In the first two targets presented there is some modest variation in the required number of traces required to see leakage as we increase the number of iterations of the bootstrapping method. However, for the third target the difference is much larger. If bootstrapping is applied 10 times we require 450 traces to detect leakage, whereas we only require 40 if bootstrapping is applied 100 times. Both of these numbers stand in stark contrast to the number of traces required by a straightforward TVLA, which is in the order of 1×10^4.

6.4 Limitations

The idea of the bootstrap technique is to get an estimate of the deviation of a sample statistic from the true value of the statistic, and relies on the independence of the samples to do so. It

does not allow one to extrapolate information from the underlying data if it is not represented in the acquired set. What it can do is give us some assurance on the test statistic and its variation to give more accurate picture. That is, if the collected data set is representative of the underlying distribution, resampling will help produce a more accurate statistical analysis. There are obviously limitations of this technique, as demonstrated in Figure 6.7. The top left plot shows the result of a straightforward fixed-versus-random TVLA test, as described in Section 6.2.1, on 5×10^5 traces, where the t-test statistic is turned into a p-value under the null hypothesis that there is no leakage. From this picture, it is clear that some points are already crossing the 4.5σ line (i.e. where $-\log_{10} p = 5$), while other points are getting close to the line. As has been clear from the literature, the results of a t-test are greatly affected by the signal-to-noise ratio of the measurements, and reliably identifying false negatives and false positives is problematic. The bottom right plot shows the bootstrapping method applied $b = 5$ times to the same 5×10^5 traces (we note recommendations on b vary widely in literature). This demonstrates that we get a lot more assurance on the points that do not provide evidence the null hypothesis is correct and all points which showed leakage in the original figure are present. The top right plot shows the result of bootstrapping a 1000 traces with b=20, and the bottom left plot shows the result of a bootstrapping of 5000 traces with b=5. Neither of these figures are showing the peak around sample point 30 visible in the top left plot indicating that the underlying data is not sufficiently representative of the full set because we restricted the amount of traces. However, we do have peaks at other points that are not visible in the entire set, again caused by bias in the smaller number of traces. While bootstrapping can allow one to determine if leakage is visible on a smaller number of traces, it is subject to bias in the acquired traces.

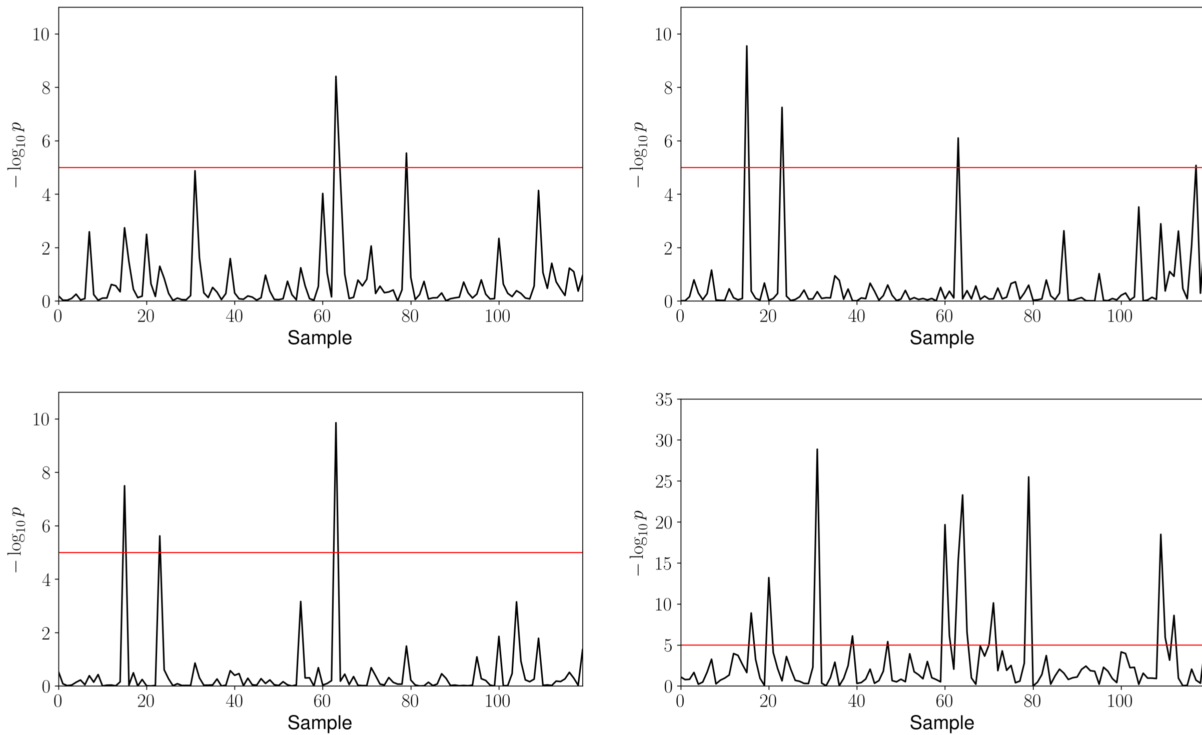

Figure 6.7: The negative log of p-value returned by the TVLA test for a fixed-versus-random t-test with 50000 traces (top left), 1000 traces with 20 iterations of the bootstrapping method (top right), 5000 traces with 5 iterations of the bootstrapping method (bottom left) and 50000 traces with 5 iterations of the bootstrapping method (bottom right)

6.5 Implementation Details

Statistical processing for side-channel analysis can be computationally intensive and, since bootstrapping runs the statistical analysis multiple times, the process can be even more demanding. The most straightforward approach to computing statistical tests is to store all the data on to a hard disk, read the measurements, run the data through the algorithm of interest and compute the results. Another approach is to use one-pass algorithms, which find the required statistical characteristics during acquisition. This concept varies from having all the statistics ready for the next update to updating an accumulator for each new sample and computing results on demand [119, 129, 134, 148].

Our method requires calculating different statistical tests (i.e., Welch's t-test and KS-test), which use statistical moments as well as observation frequencies. Hence, we chose a histogram approach, where the histogram contains all information about the distribution that becomes available while acquiring traces and as such describes the sample distributions. It is then possible to derive properties appropriate for both tests. Our statistical technique was initially based on the work by Reparaz [129]. However, our implementation extends the method to a coordinate space, provides an implementation algorithm, and describes how to calculate statistics other than the t-statistic.

We assume that the leakage assessment is performed over a set of observed samples S with n traces of m sample points with c classifications. Each sample point in the measurement has r meaningful bits, corresponding to 2^r integer values, which are used as indices of counter bins. Each classification should have q sets of histograms, where q is the amount of bins required to cover each possible classifier value. This approach can be represented as a 4-dimensional set $_{cqm2^l}$. We shall denote an instance of this set as $\mathcal{H}$. An element of $\mathcal{H}$ is denoted e_{ijkl} where $i \in \{1, \dots, c\}, j \in \{1, \dots, q\}, k \in \{1, \dots, m\}, l \in \{1, \dots, 2^r\}$. For example, if there in an evaluation of the non-specific fixed-versus-random test we have $c = 1$ and $q = 2$. If we would wish to conduct a correlation power analysis [28] on an 8-bit intermediate state with the hamming weight model we would have a separate classifier, e.g. $c = 256$, with $q = 9$.

When acquiring data one would set $\mathcal{H}$ to all zeros and update $\mathcal{H}$ after each acquisition using Algorithm 6. At any given moment, the results of the statistical tests can be rapidly computed from $\mathcal{H}$.

Algorithm 6: Updating $\mathcal{H}$

Input: $\mathcal{H}$ with elements e_{ijkl} where $i \in \{1, \ldots, c\}$, $j \in \{1, \ldots, q\}$, $k \in \{1, \ldots, m\}$, $l \in \{1, \ldots, 2^r\}$, a set of n traces $S = \{s_1, \ldots, s_n\}$ with $\sim_{\approx} = \{s_{t1}, \ldots, s_{tm}\}$ with a classifier value z_{ti} for each of the classifications. For ease of notation, classifier values will be in $1, \ldots,$ rather than the actual value.

Output: $\mathcal{H}$

```
1  for t = 1 to n do
2      for i = 1 to c do
3          for k = 1 to m do
4              j ← c_i ;
5              l ← s_{t,k} ;
6              e_{i,j,k,l} ← e_{i,j,k,l} + 1 ;
7          end
8      end
9  end
10 return H
```

The first two statistical moments, μ and σ^2, for Welch's t-test:

$$\begin{aligned} \mu_{ijk} &= \frac{1}{N_{ijk}} \sum_{x=1}^{2^r} e_{i,j,k,x} x \\ \sigma^2_{ijk} &= \frac{1}{N_{ijk} - 1} \sum_{x=1}^{2^b} e_{i,j,k,x} (x - \mu_{ijk})^2 \end{aligned} \tag{6.7}$$

where $N_{ij} = \sum_{x=1}^{2^r} e_{i,j,1,x}$. Then we can compute the t-statistic, applied to k points, as

$$t = \frac{\mu_{ij'k} - \mu_{ij''k}}{\sqrt{\frac{\sigma^2_{ij'k}}{N_{ij'}} + \frac{\sigma^2_{ij''k}}{N_{ij''}}}}. \tag{6.8}$$

.

The CDF function used to define the sampled distribution, see (6.3), used to compute the

KS test, applied to k points, becomes:

$$d_{ijkx} = \sum_{y=1}^{x} e_{i,j,k,y}. \tag{6.9}$$

We note (6.7) and (6.9) use the notation used in Algorithm 6, where i is a classifier index, j is a bin, k is a trace sample point, and x is a counter bin index.

This approach was been implemented as a python module, compiled from cython code to C code and then to a dynamically linked library. The Intel MKL library has been used to derive the required statistics. The space $\mathcal{H}$ has an element type represented by a 32-bit unsigned integer, which establishes the memory requirement for $\mathcal{H}$ as $4 \times c \times q \times m \cdot 2^b$ bytes. This would allow one to process up to 4 billion traces, which is typically sufficient. It is important to note that the size of $\mathcal{H}$ should be small enough to fit within CPU cache, which is typically 5, 7 or 15MB. This implementation allowed us to efficiently evaluate the Bootstrapping method.

6.6 Conclusion

In this chapter, we describe how to use bootstrapping to augment side-channel leakage detection tests by repeated sampling with replacement from an acquired set of traces and combining the results of each set. Simulations and experiments show that even a small number of iterations of the bootstrapping method present significant improvements over straightforward TVLA [67]. The bootstrapping method presented above can be applied to other statistical leakage detection methods [30, 110], and we would likewise expect a similar increase in performance at the cost of extra calculation time. We also show an efficient way of computing the necessary statistics to compensate for the extra calculation time, based on

methods described by Reparaz [129].

Recent work by Bache [15] proposed a somewhat similar approach to our work, although without the application of bootstrapping. They describe using the confidence interval, instead of a single p-value/t-statistic, to improve the assurance of the presence, or absence, of leakage. The confidence interval provides the error-probability for a false negative. However, the confidence interval makes it harder for an evaluator to make a judgment about leakage, when compared to the pass/fail criteria used in straightforward TVLA. In comparison, applying bootstrapping to TVLA, as we describe, provides a single pass/fail parameter from combining p-values, making the results easier to interpret than those provided by the method presented by Bache [15]. Moreover, since applying bootstrapping extracts more information from an existing set of acquisitions, applying bootstrapping to TVLA improves the data-efficiency in leakage detection. That is, it can predict/detect leakage with fewer acquisitions. In comparison, the accuracy of the method presented by Bache using the confidence interval is highly dependent on the number of acquisitions.

Chapter 7

Programmable RO (PRO): A Multipurpose Countermeasure against Side-channel and Fault Injection Attacks

In this chapter, we will present our design of a multipurpose secure primitive that can proactively monitor and protect the security of on-chip Power Distribution Network (PDN).

7.1 Introduction

In a physical side-channel attack, an adversary learns secret information by passively monitoring or else actively influencing the implementation of a secure electronic system. While power consumption is a popular target in side-channel attacks, many other sources of physical quantities have been identified and used as side-channel leakage. Besides passive monitoring of circuit behavior, an additional cause of information leakage stems from targeted faults. By analyzing the corresponding fault response, an attacker can retrieve the secret information from a target [18]. The most common methods to inject faults include power glitches, clock glitches, electromagnetic pulses, and laser pulses. Finally, fault injection and side-

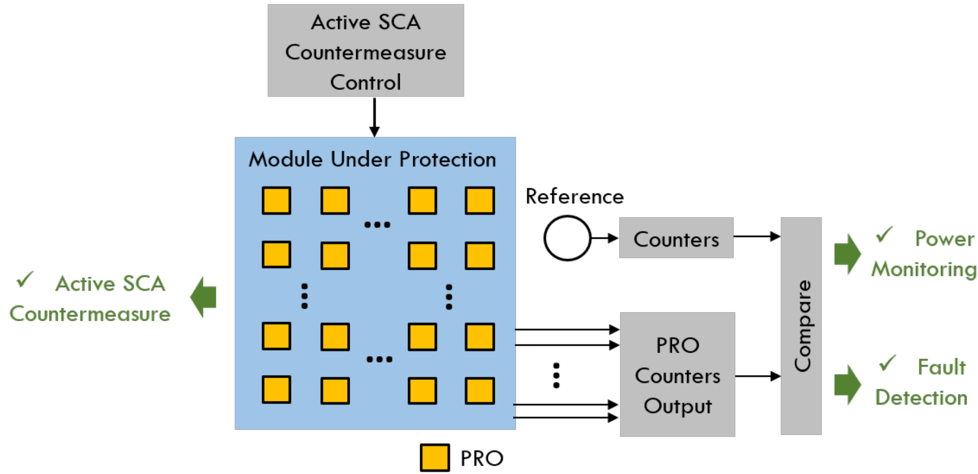

Figure 7.1: PRO based on-chip Secure Network Hardware Extension

channel monitoring can also be used in a combined attack, for example, to break a masking side-channel countermeasure [153].

Even though many existing works have demonstrated side-channel and fault attack countermeasures, there are no simple circuit-level solutions to solve **both** side-channel and fault attack vulnerabilities in a generic manner. Generally, even for individual side-channel or fault countermeasures, a significant overhead will be introduced to the design. Moreover, many of the existing countermeasure mechanisms have to be specifically adjusted for the implemented algorithm.

In recent years, researchers have further demonstrated that the placement of the attacker and the victim circuitry on the same chip while sharing a common PDN brings new side-channel and fault attack opportunities. Having a common PDN intrinsically relates the perturbations from the victim's logic to the attacker's logic and vice versa. Therefore, a neighboring adversary logic can interpret information about the victim operations by monitoring the changes on the shared PDN. On the other hand, the same physical effect exists in the other way around; The victim logic can infer malicious operations of its neighbor circuitry

by monitoring the shared PDN. Therefore, in order to guarantee the security of the PDN, a monitoring sensor network on the PDN should be built to detect on-going attacks. The monitoring sensor network should fullfill the requirements including large spatial coverage, i.e., covering the full PDN area, and large temporal coverage, i.e., continuously monitoring the PDN [140].

Previously, Ring Oscillators (ROs) were widely used by silicon design houses as test structures or on-chip sensors to monitor the performance of their technologies and circuits [82]. But a multi-purpose design of RO-based on-chip sensors has not been investigated in adding resistance against both side-channel and fault attacks to the circuit. In this work, we introduce a new multi-purpose Ring Oscillator design - programmable RO (PRO). With a low overhead, the proposed PRO can provide the following solutions within the same structure:

- Active Side-channel hiding countermeasure;
- On-chip power monitoring;
- Fault injection monitoring;

The proposed PRO design has multiple configurations of oscillation frequency, which are under the control of the user (i.e., the defender). Each PRO has its own counter which can be read to calculate the PRO's frequency by comparing it with a reference counter. We first demonstrate that with low overhead, an individual PRO can provide sufficient disturbance to the power to hide the side-channel leakage of the secret information in the system. Moreover, we further demonstrate that by combining multiple PROs into an array and by placing them within the module under protection, a secure on-chip monitoring network can be constructed to monitor the power fluctuations on the PDN to detect abnormalities and fault attacks.

Figure 7.1 shows the overall structure of the PRO-based on-chip secure system. The PROs

are evenly placed on the chip to form a secure on-chip network. The PRO secure network can be controlled by the external user configuration. The user can turn on the SCA countermeasure by configuring the PRO to oscillate at randomized oscillation frequencies. Besides, the user can monitor the oscillation frequency of each PRO in the array by reading out its corresponding counter value. We demonstrate that by monitoring the frequency change of the PROs, on-chip local power attacks and EM fault injections can be detected.

The proposed design can be used on any secure module, from small hardware accelerators to complex System-on-Chips (SoCs). To the best of our knowledge, this is the first work to comprehensively study the potential of RO-based designs in SCA countermeasure, power sensing, and fault detection.

Adversary Model

PRO covers adversaries with side-channel and fault attack capabilities listed in the following:

Side-channel Attacker Model. We consider two attacker models. The first attacker model has physical access to the device which enables the attacker to control the input data and monitor the power dissipation by shunting the device's power supply. The second attacker model works remotely; The attacker circuit shares a PDN with the victim circuit and can control *only the attacker circuit* remotely. Therefore, the attacker is able to implement malicious logic to monitor the changes on the shared PDN and measure the power consumption of the device [139, 162]. This enables the attacker to perform side-channel attacks, such as Simple Power Analysis (SPA) [101], Differential Power Analysis (DPA) [85], and Correlation Power Analysis [29], to retrieve the secret information used in the victim circuit.

Fault Attacker Model. We also assume the adversary can induce faults into the victim

circuit by stressing the electrical environment, such as injecting clock glitch, power glitch, and EM glitch. These glitches can induce targeted transient faults which can flip bits, change the control flow of the secure algorithm, set/reset the circuit, etc. Fault injection can be done either by exerting disturbance to the circuit directly, which requires the adversary to have physical access to the device, or by having remote access to the shared cloud computing environment with the victim circuit [10, 89, 96, 126]. The exact fault effects to the circuit highly depends on the fault injection parameters, victim circuit's architecture and algorithm, and fault injection technique. By monitoring the fault response of the circuit after injecting targeted faults, the adversary can retrieve the secret information by performing Differential Fault Analysis (DFA) [23], Statistical Fault Analysis (SFA) [60], or instruction skip attacks [152]. PRO as a secure on-chip add-on can be integrated to the circuit to protect against the aforementioned attackers. Adversaries may try to tamper with the PRO sensor itself to bypass the PRO's security mechanisms, but we don't consider this adversary model within this work.

The structure of this chapter is as follows. The next section reviews related work of Ring Oscillators and highlights our contribution. section 7.3 describes our proposed PRO design. In section 7.4, we explain and demonstrate the effectiveness of PRO as a side-channel countermeasure. Next, we present the PRO's power sensing functionality in section 7.5. We further show that PRO can detect power fault and EM fault in section 7.6. Finally, we conclude this chapter in section 7.7.

7.2 Related Work

When sharing the same PDN, seemingly unsuspecting parts of the implemented logic can perform adversarial operations on the other parts. In this work, our focus is on two categories

of adversarial operations; fault injection and power side-channel analysis. In the following, we categorize the related work into three parts: using on-chip logic as a countermeasure against power SCA, using on-chip sensors as power sensors to detect power perturbation, and using on-chip sensors to detect fault injection attacks.

7.2.1 On-chip sensors as a countermeasure against power SCA

Liu et al. [94] use an array of ROs, randomly switched on and off, to dynamically hide the power consumption of AES SBox and hinder the first-order Differential Power Analysis (DPA). Similarly, Krautter et al. [90] use ROs as a power-based SCA mitigation methodology. In their work, the part of the implementation that needs to be protected is surrounded by a network of ROs. By switching an arbitrary number of the ROs on and off, the Signal-to-Noise Ratio (SNR) in power traces decreases, and therefore, the number of traces required for a Correlation Power Analysis (CPA) attack to be successful is increased. This approach is called *hiding* side-channel leakage. However, the RO in both designs are running at a fixed oscillation frequency, and thus, only a single-frequency noise is injected. In this case, it is straightforward for an attacker to apply post-processing techniques to remove the noise effect. To avoid this weakness, PRO uses user-controlled but random frequency changes (section 7.4). Moreover, to further reduce the overhead, we show how a simple modification can enhance the countermeasure efficacy.

7.2.2 On-chip sensors to detect/cause power perturbation

Zick et al. [163] use ROs to measure on-chip voltage variations. Indeed, the oscillation frequency is proportional to the supplied voltage on the PDN. To measure the frequency

of an RO accurately, counters are required that are clocked with the output of the RO. This limits the maximum sample rate attainable by the RO counter structure, and hence, the bandwidth of the side-channel signal. This limitation has motivated research on other voltage-sensitive Time-to-Digital Converter (TDC) methods. For instance, Gnad et al. [66] use carry-chain primitives available on Xilinx FPGAs as TDCs. However, the use of carry-chain primitives makes their approach specific to certain FPGA families. Similar TDC structures have been explored in the context of CMOS design simulation to measure the operating voltage of a chip [14].

Moreover, ROs have been used in offensive scenarios affecting the PDN for both passive (power-based) and active (fault injection-based) physical attacks. As an example of power-based SCA, Zhao et al. [162] presented on-chip power monitors with ROs. They demonstrated that ROs can be used as a power monitor to observe the power consumption of other modules on the FPGA or SoC. Using their power monitor, they captured power traces of the device running the RSA algorithm and were able to successfully find the private key by applying Simple Power Analysis (SPA). Gravellier et al. [69] perform CPA on power traces acquired with RO-based power sensors.

To inject timing faults, Mahmoud et al. [96] employ ROs to increase the voltage drop on the power network and lower the voltage level. Effectively, they make the victim chip slower, causing timing faults. Similar attacks have been shown in other works [10, 89, 126].

7.2.3 On-chip sensors to detect fault injection

Next, we consider on-chip sensors for fault detection. Miura et al. [105] present a sensor consisting of Phase-Locked Loop (PLL) and ROs. In their work, ROs are routed in a specific way to ensure their path travels through most parts of the chip. Once an EM fault is injected,

the path delay of the ROs will be affected, resulting in changes in the RO phase. The PLL logic can capture this phase disturbance and detect the ongoing fault injection. Similarly, He et al. used PLL block to detect the laser disturbance on RO oscillation frequency [73].

Provelengios et al. [126] show that on-chip ROs can not only detect fault injection, but also locate the origin of the fault injection. With a similar structure, RON [161] builds a ring oscillator network, distributed across the entire chip, to detect hardware Trojans. Their work confirmed that RO-based power sensors can have a sufficiently high sample rate to detect fluctuations on the PDN.

However, the scope of their work is limited to the power fault detection, whereas, in our work, we further investigate EM fault detection (section 7.6). Additionally, the unique programmable design of our proposed RO structure also enables its usage for power SCA countermeasure (section 7.4).

7.2.4 Our contribution

In general, each previous work addresses one single aspect at a time: a side-channel countermeasure, a power monitor, or a fault detector. In practice, an adversary is capable of performing a combination of attacks. Hence, it is crucial to find a security mechanism that encapsulates protection against these attacks. Our goal in this work is therefore to design a *programmable* RO structure that can provide the following functionalities within the same structure:

1. Hiding protection against power-based SCA.
2. On-chip power monitoring of the fluctuations on the PDN.
3. Detecting fault injection.

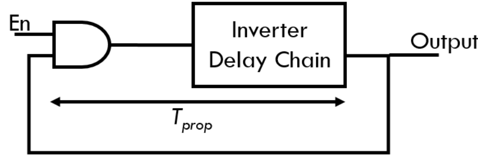

Figure 7.2: Propagation delay of a ring oscillator.

To the best of our knowledge, this is the first work to comprehensively investigate the RO's potential in addressing all these three aspects. In the following sections, we introduce our proposed design and demonstrate through experiments the capability of the proposed system. Even though we demonstrate our experiments as an FPGA prototype, our design is not limited to FPGAs and can be extended to other electronic chips.

7.3 Programmable RO Design

7.3.1 Background

In this section, we introduce our Programmable RO (PRO) sensor design. As shown in the Figure 7.2, Ring Oscillator (RO)'s output oscillation frequency depends on the propagation delay of their internal signals. In each oscillation period of a RO, the signal has to propagate twice through the propagation path. Therefore, the oscillation period (T_{RO}) of a RO is $T_{RO} = 2 \cdot T_{prop}$ and its frequency follows the following equation:

$$f_{RO} = \frac{1}{2 \cdot T_{prop}} \tag{7.1}$$

More specifically, the propagation delay path is composed of an odd number of inverters and each inverter contributes to the delay of the path. If t represents the delay of an individual inverter, and n denotes the number of inverters in the chain, its frequency follows

the following equation:

$$f_{RO} = \frac{1}{2n \cdot t} \tag{7.2}$$

Hence, the frequency of a RO can be controlled by adjusting the number of stages in the inverter chain.

7.3.2 PRO Design and Configuration

In this work, we aim to have a programmable design of the RO which gives the designer the flexibility to choose the RO oscillation frequency.

Figure 7.3 shows the basic structure of our proposed design of the programmable sensor. The PRO consists of multiple delay cells. Each delay cell includes two delay paths; one consisting of inverters and the other a shorting path which bypasses the inverters. The multiplexer in the delay cell can control the delay cell's propagation delay by selecting between the delay path and the shorting path with the control input signal `SEL`. Each delay cell has its independent control signal. Suppose there are N inverters configured in the delay cell, when `SEL` $= 1$, the delay path is selected and when `SEL` $= 0$, the shorting path is selected. The propagation delay of each delay cell T_C is therefore:

$$T_C = SEL \cdot T_d + (1 - SEL) \cdot T_s \tag{7.3}$$

Where T_d denotes the propagation delay of the delay path and T_s denotes propagation delay of the shorting path. The propagation delay of the shorting path T_s is a very small value compared to T_d but not 0, this is because of the delay of routing and the delay of the multiplexer. Other user control inputs include `EN` which controls whether PRO is enabled (oscillating) or not, and a control signal to reset/read the PRO counter. The structure

of the PRO design gives flexibility to the designer in manifold. As shown in Table 7.1, there are multiple initial structural configurations to be decided by the hardware designer at design time, including the number of inverters per delay cell, as well as the number and type of different delay cells. These parameters determine the range of the programmable RO's oscillation frequency and the number of frequency configurations the programmable RO can have.

Several constraints can be used as the guidance while configuring the Initial Design Configurations of PRO:

1. Oscillation frequency range;
2. Number of configurations;
3. Size of frequency changing step;
4. Area;

As a starting point of PRO parameter configuration, the designer should estimate the propagation delay T_{prop} for a single inverter. This knowledge can be obtained through the design library, timing simulation, or measuring RO's oscillation frequency with a single inverter (when working on an FPGA environment). Then, based on the designated Oscillation frequency range of PRO, the designer can calculate the minimum and maximum number of inverters are needed by Equation 7.2. After deciding the number of inverters needed, the designer can group the inverters into different types of delay cells based on the designated frequency changing step and number of configurations that are needed. Theoretically, more inverters result in a larger oscillation frequency range at a cost of larger PRO area. Therefore, based on the targeted protect design area, the designer should decide the area constraint for the PRO, so that the each PRO can have a good spatial coverage of the design while at the

Table 7.1: Configurations for PRO

Configuration Type	Configurations
Initial Design Configurations	number of delay cell type, number of delay cells, number of stages in delay cells
User Configurations	`EN`, `SELs`, Counter Start/Stop

same time wouldn't be too close to influence other PROs' local power distribution.

Next, to better explain our proposed structure, we pick one configuration as an example. Figure 7.3 shows the structure of the PRO with three types of delay cells. The type-0 delay cell (D0) has 4 inverters, the type-1 delay cell (D0) has 8 inverters and type-2 delay cell (D0) has 16 inverters. We instantiated 2 of each type of delay cells in the inverter chain. All the delay cells have an even number of inverters, and 1 inverter is instantiated at the start of the inverter chain to make sure that there is always an odd number of inverters in the inverter chain. When all the inverters are configured to be used in the delay path, the propagation delay T_{prop} is maximal, therefore, the overall programmable RO will oscillate at its lowest frequency. When all the delay cells are configured to use the shorting path, the propagation delay T_{prop} is minimal, therefore, the overall programmable RO will oscillate at its highest frequency.

In our experiment setup, we implement PRO on Xilinx Spartan-6 FPGA, which is fabricated with 45nm CMOS technology. Under the aforementioned configuration, we measured that the lowest oscillation frequency is 22MHz and the highest oscillation frequency is 123.44MHz. Since each delay cell's `SEL` is independent, there are in total 15 frequency configurations consisting of {1, 5, 9, ..., 57} inverters. Since there are six `SEL` signals, there are 64 configurations in total which redundantly map into the 15 achievable configurations. Through this redun-

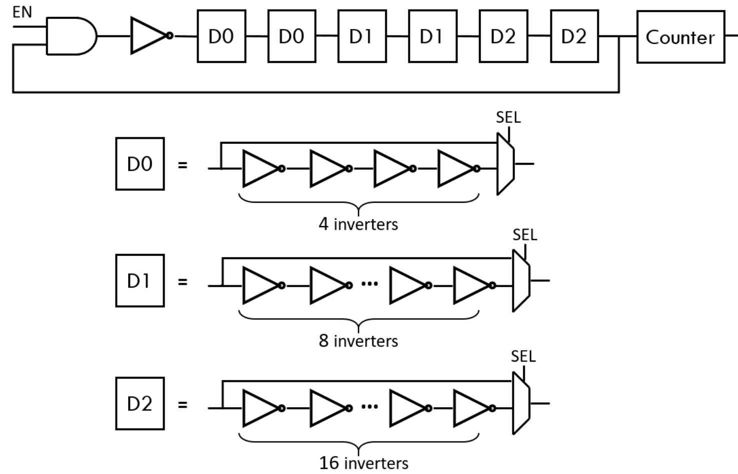

Figure 7.3: PRO Design. D0 donates the delay cell type-0, D1 donates the delay cell type-1, D2 donates the delay cell type-2.

dancy, we are able to estimate the local manufacturing process variations, which is helpful to decide when a deviation should be cause for alarm (i.e., fault detection) or not.

The designers can control the RO's frequency by setting the input value of `SEL`. We are using the same configurations for all the later experiments in this chapter. For under this PRO configuration, each PRO can be implemented with 128 LUTs and 32 Registers, in total 160 slices. In our experimental setup, a PRO array with 36 PROs can cover the whole FPGA (46648 LUTS and 93296 Registers, in total 139944 slices) to provide the whole chip power monitoring and fault detection. Therefore, only an overhead of 4.1% is introduced.

7.3.3 PRO Integration and Basic Principles

As a security resistance add-on, PRO can be integrated into the design to protect simple designs such as hardware encryption engines as well as complex systems such as an SoC. Control signals are needed for communicating with the PRO. The control signals set up the user control configurations in Table 7.1. Generally, different control mechanisms can be

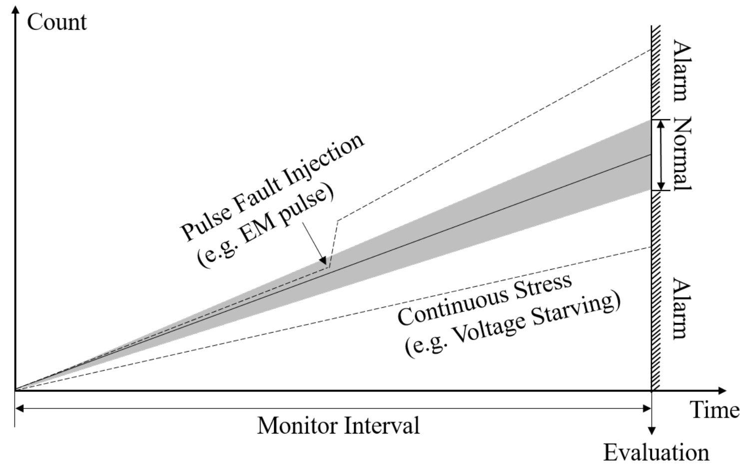

Figure 7.4: Basic principles for PRO fault detection

adopted by the designer. In an SoC, the designer can add PROs as a co-processor which can be controlled by the processor through memory-mapped registers. Under this environment, the software running on the processor can configure the PROs on the chip. Therefore, PRO-based countermeasures can be dynamically enabled/disabled while the software is running. Besides, a hardware-based Finite State Machine (FSM) can be used to control the PROs as well. In our experiments, we are using the UART protocol to communicate with PRO, and the control signals sent through the UART are generated by a python script in this chapter.

Figure 7.4 shows the high-level basic principles for the fault detection mechanism using the PROs. The counter value will be evaluated at the end of each monitoring interval and compared with the reference counter value to get the actual oscillation frequency of the PRO. Under normal circumstances, each PRO oscillates at a certain constant frequency, and thus, its counter value will increase linearly during the monitoring interval. There will be some small variances caused by the environmental changes, jitters, process variance of the manufacturer, etc. A characterization procedure, therefore, is needed to define the range of normal operation [140]. However, in the occurrence of instant fault injection (e.g. power glitch, EM pulse, time glitch, laser pulse), the counter will be disturbed. The counter value

read out at the evaluation time will deviate from the normal range, and thus, a pulse fault injection will be detected by the PROs. Additionally, an adversary can inject timing faults by stressing the PDN continuously (e.g.power starving). As a result, the victim circuit will operate slower and cause timing violations to create faults. In this case, the PRO counter value will also deviate from the normal value and capture the fault injection event. In this chapter, we use power fault and EM fault as an illustration, but PRO's fault detection coverage is not only limited to these two fault types.

7.4 Side-channel Countermeasure

Masking and hiding are two popular techniques for side-channel countermeasures. In masking, each secret variable is split into two or more shares which are concealed by random numbers. The side-channel leakage of each share alone does not reveal the secret variable because of the randomization introduced by random numbers. A random source that provides fresh random variables is significantly important in masking implementations. Hiding countermeasures reduce the SNR for secret data-dependent operations. Hiding can be achieved by several techniques, such as by reshuffling cryptographic operations in a data-dependency consistent but random order [141], inserting random delays [36], and running multiple tasks in parallel [139]. In this work, we utilize the proposed PRO design as a hiding countermeasure by injecting noise with random frequency. Previous work has proposed injecting noise for reducing the SNR [39] [95]. However, since only single-frequency noises are injected, it is not tricky for an attacker to decrease the effect of noise either by using a band-pass filter while collecting traces or by post-processing the collected power traces, such as applying averaging, filtering, and frequency domain analysis. Thus previously proposed noise-injection-based hiding mechanisms still have security flaws. In our proposed design,

we inject random-frequency noises with the PRO design so that it will be much harder for an adversary to eliminate the noise.

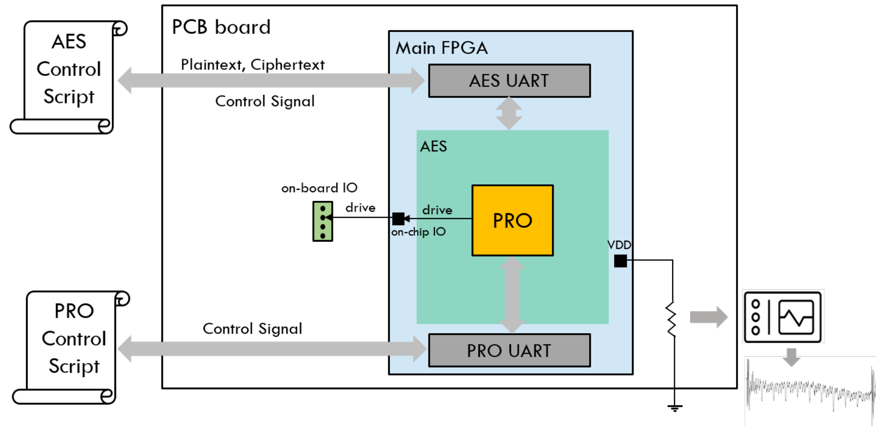

Figure 7.5: Experimental Setup for Evaluating RO's performance in side-channel leakage hiding

The countermeasure circuit consists of a single PRO whose frequency can be controlled by the `SEL` input signals. The PRO drives one of the IO pins on the board. As demonstrated in previous work [95] [91], the power consumed by a single RO is not large enough to have a significant influence on the power profile of a complete chip or a complete cipher. Instead, hundreds of RO need to be instantiated on the chip to have a profound hiding influence. This approach will cause significant design overhead and has the potential risk of inducing power fault to the circuit [84]. In our proposed mechanism, by driving an I/O pin with a PRO, the effect of a single (randomly-switched) PRO to influence the off-chip power network is amplified. Since the load capacitance of an IO-pin is much larger than the load capacitance of an internal FPGA net, the IO-pin requires more energy to charge and discharge and the overall power spectrum will thus be influenced by the oscillation frequency of the RO. In this manner, even with a single Programmable Ring Oscillator (PRO), significant additional power is consumed to change the power consumption characteristic. In practice, an adversary senses the on-chip power consumption using a probe, either by connecting an external probe

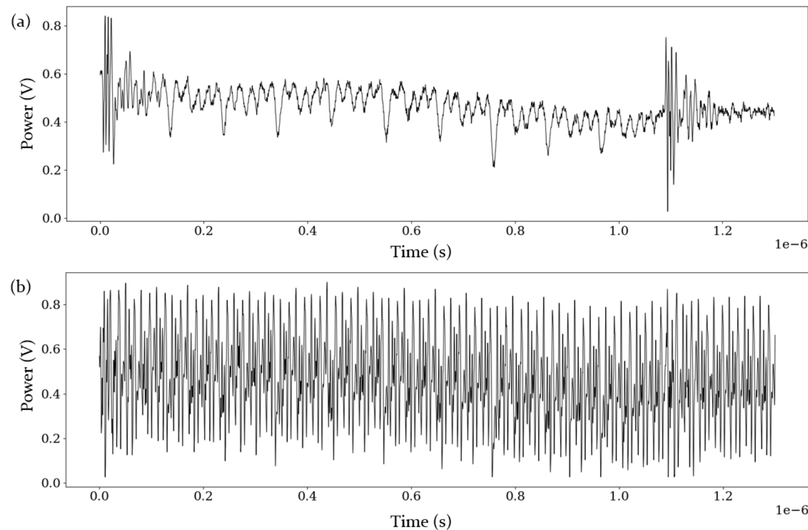

Figure 7.6: AES power traces when PRO is (a) Off; (b) On;

to the system via a power supply pin [146] or else using an EM probe. Both of these are dependent on the off-chip power network, and therefore, affecting the off-chip power network is an important factor to defeat an attacker maliciously monitoring the power profile.

The performance of our proposed hiding countermeasure design is evaluated with AES. Figure 7.5 shows our experimental setup. We put hardware AES as well as the programmable sensor on the FPGA. The output signal of the PRO is mapped to drive the IO pin to amplify the noise effect. For each encryption scenario, plaintext and ciphertext are provided through the UART for AES. The communication procedure is controlled by the AES control script. At the same time, we use the sensor control script to send in control signals through the PRO UART. The control signals can enable/disable the RO and configure the oscillation frequency of the RO. While AES is running, the sensor's control script generates random numbers for the frequency configuration so that the frequency of the PRO can change randomly. Equally, an on-chip Pseudo-Random Number Generator (PRNG) can be used for this purpose. Figure 7.6(a) shows the collected AES power trace when the programmable sensor is off. We can clearly see the pattern of ten rounds of the AES algorithm. By com-

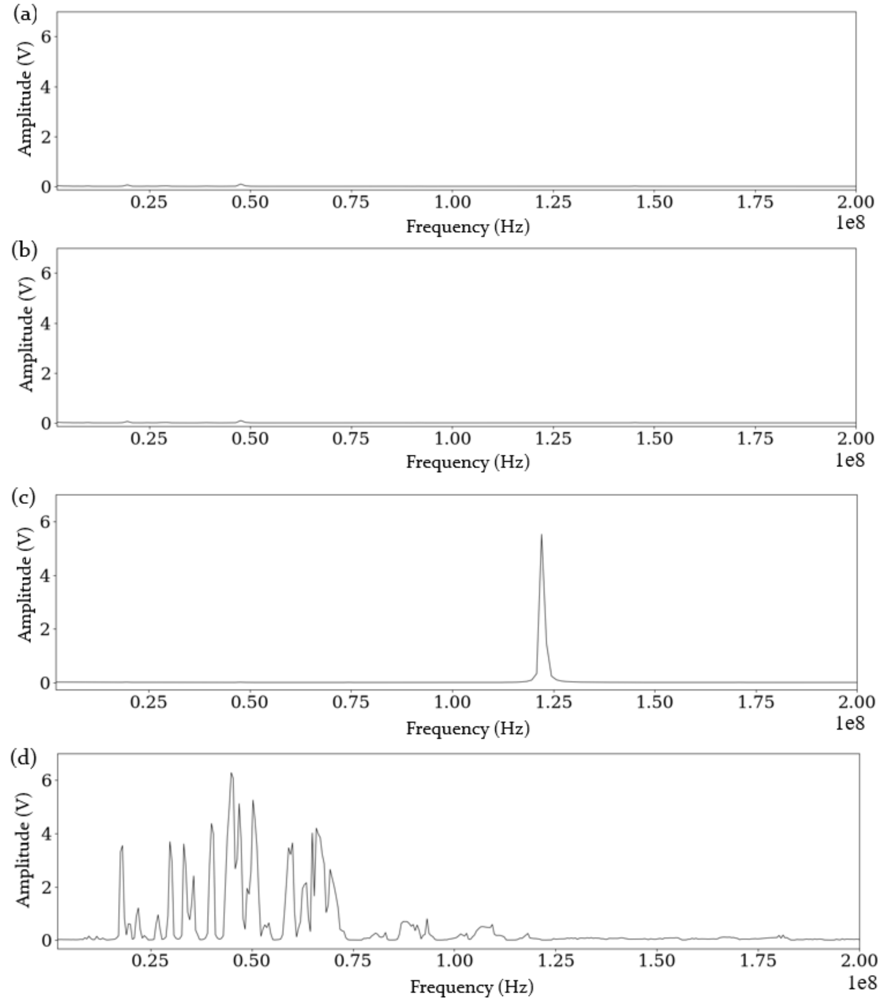

Figure 7.7: Power Spectrum for power traces when (a) PRO off; (b) PRO on without driving IO pin; (c) PRO with fixed oscillation frequency and driving IO pin; (d) PRO with random oscillation frequency and driving IO pin;

parison, the power trace changes to a repeated oscillation pattern when we turned the PRO on, as shown in Figure 7.6(b), which indicates the strong influence of PRO on the power profile. Under our setup, the complete AES takes 41ms, and we configure the PRO control script such that the frequency of the PRO changes every 2ms, which means that the PRO's frequency will change at least 20 times while AES is running. Figure 7.7(a) shows the frequency spectrum of the power traces when the PRO is off. We can observe small peaks at the clock frequency (24MHz). We do not observe a significant influence on the power spectrum

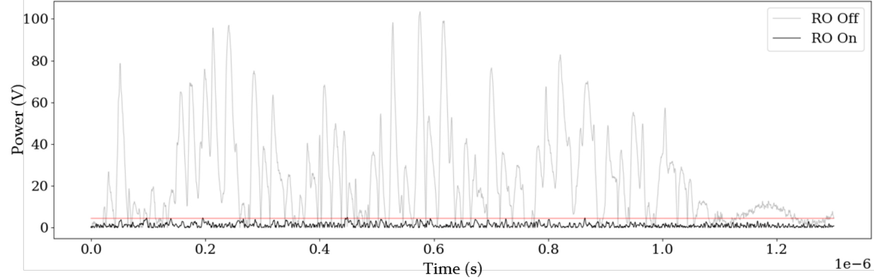

Figure 7.8: T-value Comparison when PRO is on/off

if we only put a single PRO without driving the IO pin. Figure 7.7(c)(d) shows the power spectrum when the PRO is on and driving the output pin. By comparing to Figure 7.7(a), a significant influence on the frequency spectrum of the power profile can be observed while PRO is on. Figure 7.7(c) shows a sharp peak when we fix the PRO's oscillation frequency to 120MHz.

Suppose one tries to protect the secure component by injecting noise with a regular RO with a single oscillation frequency. It is easy for the attacker to implement frequency spectrum analysis, find the injected noise frequency, and apply the corresponding filter to eliminate the influence of the injected protection noise. As a sharp comparison, Figure 7.7(d) shows that when random frequency noise is injected by PRO the frequency spectrum is expanded within the PRO's oscillation range from 22Mhz to 123.44Mhz. This makes it much harder for the adversary to filter out the noise by post-processing. To further evaluate the effectiveness of the proposed design on increasing side-channel resistance, we applied TVLA [67] on 50k collected traces; As shown in Figure 7.8, a dramatic decrease of t-value can be observed when the PRO is turned on compared to when the PRO is off. This indicates that the PRO design can significantly reduce the side-channel leakage of the circuit. We calculate the average power value with PRO countermeasure off is 0.4643W and with PRO countermeasure on is 0.4692W, therefore, the overhead of power consumption is 1%.

Note. Generally, even though the adversary is aware of the noise signal, since the noise

is injected by PRO at a random frequency which also changes at a fast pace, it is exceedingly hard to remove its effect by normal post-processing techniques; The adversary needs to monitor both the power consumption and the output of the PRO simultaneously with sufficient precision and should be able to remove the part of power consumption related to the output pad's oscillation using noise-cancellation techniques, which requires high-end devices. Additionally, to have sufficient information and perform a successful side-channel analysis from the obtained side-channel traces, the sampling rate for side-channel attacks has to be at least 2× the clock frequency (according to the Nyquist theorem). We suggest that while choosing the initial design configurations in Table 7.1, the designers should adjust the configurations such that the oscillation frequency range of the PRO covers at least 3× the clock frequency. Under our experimental setup, the clock frequency is 24MHz. Therefore we configured PRO's oscillation frequency to 22MHz - 123.44MHz, which covers about 5× the clock frequency. As a result, the adversary will need a higher-end device with a much higher sampling frequency (at least 10× the clock frequency) to successfully apply the same side-channel attack. Hence, PRO as a hiding countermeasure makes it much harder to attack the circuit by largely elevating the technique bar for the adversaries.

7.5 Power Sensing

In this section, we demonstrate the on-die power monitoring functionality of the proposed PRO design. Power integrity is essential to guarantee the nominal function of the circuit. Therefore, monitoring of the fluctuations on the PDN is critical to detect abnormalities. We first explore the PRO's oscillation frequency with regard to the external power deviation. Then, we look into the PRO's performance in terms of local power sensing on the PDN of

the chip.

Electric circuits use PDNs to deliver power to the transistors in the circuit via external voltage regulators. PDNs are still affected by sudden current consumption changes despite these voltage regulators. Thus, the sudden change in the switching activity induces transient voltage drops in the PDN. PDNs can be modeled using RLC networks. The transient voltage drop seen by the PDN can be defined as follows

$$V_{drop} = IR + Lit \tag{7.4}$$

Here, the IR drop is due to the resistive components of the PDN and is dependent on the steady-state current I. The other term, *Lit*, influences voltage drop due to the inductive components of the PDN and is dependent on the changes in the current over time. Hence, as soon as there is a high current consumption/variations due to some switching activities of the logic circuit, the voltage drop will increase.

The propagation delay of signals is affected by the on-chip voltage level; Higher voltage levels increase the switching speeds of transistors, whereas lower voltage levels decrease them. Since the voltage level affects the propagation delay of signals, the immediate frequency of a ring oscillator can indicate the level of the voltage on a chip. We take advantage of this property in our proposed PRO sensor to monitor the integrity of the on-chip power network.

7.5.1 PRO Power Sensing with Regard to External Power Variations

We first investigate the PRO's frequency with respect to external power variations. Figure 7.9 shows the setup for this experiment scenario. We put a single PRO sensor on the FPGA. For the PRO's frequency measurement, we start the PRO sensor's counter and system clock counter at the same time. After running for an arbitrary amount of time T_{arb}, we read out

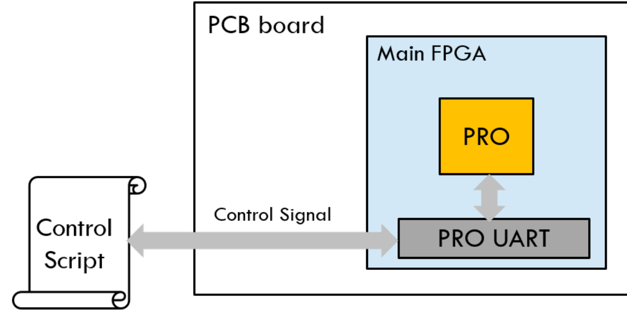

Figure 7.9: Experimental Setup for PRO frequency changing as a function of external power supply

the RO sensor's counter value C_{RO} and the reference system clock counter value C_{clk} through UART. Then, we calculate the PRO's frequency by:

$$f_{PRO} = \frac{C_{PRO}}{C_{clk}} \cdot f_{clk} \tag{7.5}$$

Where f_{PRO} is the PRO sensor oscillation frequency, f_{clk} is the reference clock frequency. We measure the value of C_{PRO} 1000 times and take the average for better precision. The measurement procedure is automated through a control script running on a PC.

As we mentioned in section 7.3, the PRO's oscillation range is 22 Mhz to 123.44 Mhz. To investigate the PRO's power sensing sensitivity when operating under different frequencies, we set the PRO sensor to several oscillation frequencies at the starting (highest) power supply voltage for the main FPGA core (1.33V). The frequency configurations we pick are 153.2MHz, 100MHz, 66.8MHz, 40.5MHz, and 27.2MHz. We gradually decrease the FPGA's supply voltage and monitor the PRO sensor's oscillation frequency.

Figure 7.10 shows the result of the PRO oscillation frequency with regard to the external supply voltage. As shown in the figure, when the external supply voltage drops, the PRO's frequency drops steadily. The PRO's oscillation frequency reflects the power supply voltage, and therefore, it can sense the changes of the power supply and can be used for power

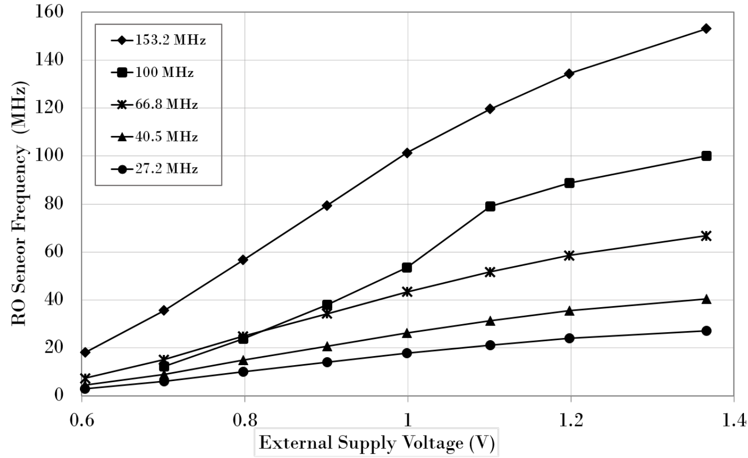

Figure 7.10: PRO's oscillation frequency with Regard of External Power Supply Voltage

monitoring. With respect to the sensitivity of power sensing, it can be observed that the higher the oscillation frequency is, the sharper the slope of the frequency vs. the external supply voltage line will be. This indicates that a higher oscillation frequency can achieve higher sensitivity in detecting power variations.

7.5.2 PRO Power Sensing with Regard to On-die Local Power Variations

After investigating the correspondence between the PRO sensor's oscillation frequency and the variations of external power variations, we evaluate the power sensing performance with regard to the on-die local power changing. Several previous works have shown that RO-based power wasters can cause a local power supply drop [106] [125] [162]. This will cause the local circuit's logic to operate at a lower voltage, therefore the local power sensor should show a decrease in the oscillation frequency when the power wasters are turned on. In this work, we adopt the RO-based power waster shown in Figure 7.11. Each power waster has five inverters in the delay chain with an `AND` gate and oscillates at 245MHz. A global enable

signal is used to turn on/off all the power wasters in the circuit.

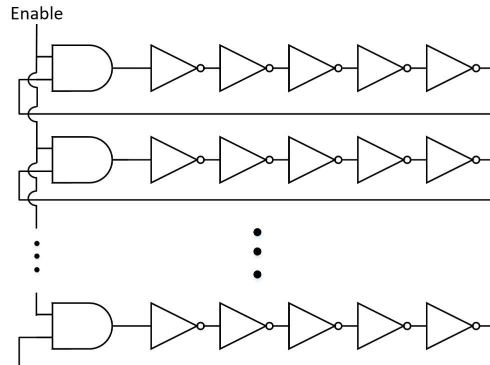

Figure 7.11: The structure of the employed RO-based power wasters.

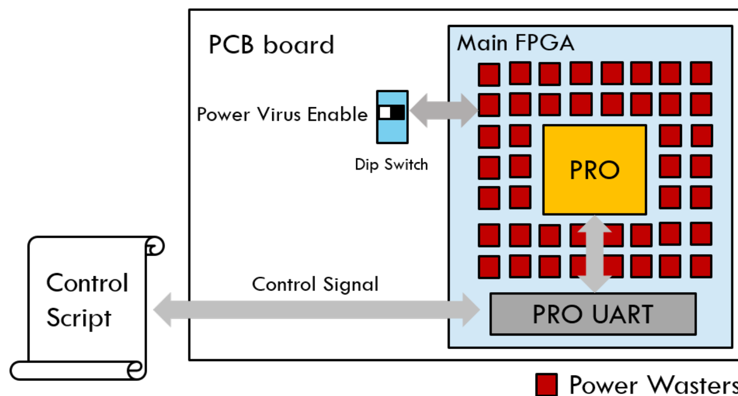

Figure 7.12: Experimental Setup for PRO Power Sensing with Regard to On-die Local Power Variations

Figure 7.12 shows the experimental setup for the local power sensing evaluation. In this setup, UART communication is used to read out PRO's counter value. We constrain the power waster to locate around the PRO sensor to induce the local power drop around the sensor. By configuring the number of power wasters, we can control the amount of local power drop. An on-board dip switch is used to enable/disable the power wasters. In a measurement scenario, we gradually increase the number of power wasters. For each number of power waster configuration, we measure the PRO's oscillation frequency 1000 times and take the average with power waster on/off, respectively. Next, we calculate the Frequency

Drop Ratio as follows:

$$Frequency\ Drop\ Ratio = \frac{f_{off} - f_{on}}{f_{off}} \tag{7.6}$$

In Equation 7.6, f_{off} denotes the PRO sensor's frequency when the power wasters are disabled (turned off) and f_{on} denotes its frequency when the power wasters are enabled (turned on). The results from the experiment are shown in Figure 7.13 when different numbers of power wasters are enabled. As more power wasters are enabled, the frequency drop ratio increases correspondingly. We can observe a nearly linear relationship between the number of power wasters and sensor oscillation slowdown. The linear regression which can closely model the correlation between the number of power wasters and the frequency drop ratio can be constructed as $f(x) = 0.00031x + 0.247$ with an R-squared value of 0.991. Therefore, we conclude that PRO can effectively sensing the local power variations as well.

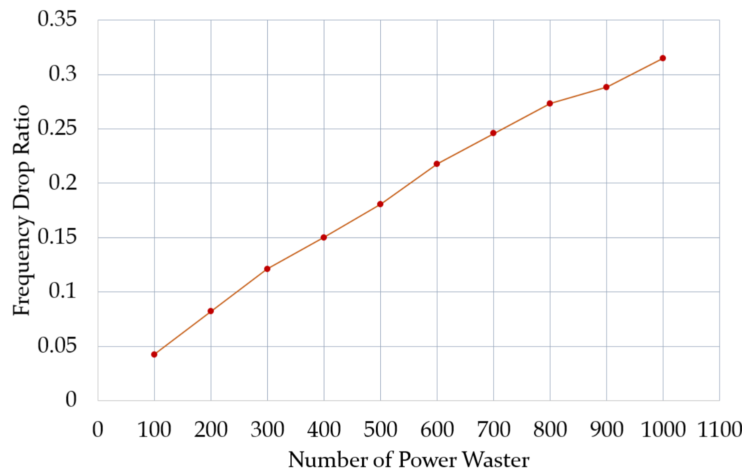

Figure 7.13: PRO Frequency with Regard of Local Power Supply

7.5.3 PRO Power Sensing with Regard to Sensor Locality

In this experimental scenario, we evaluate the PRO sensor's frequency change with respect to the spatial proximity to the switch logic that consumes the power. In this experiment,

we instantiate 36 PRO sensors to get full spatial coverage of the FPGA. As shown in the floorplan in Figure 7.14 for this experimental scenario, 36 sensors residing in 9 rows, and each row has 4 sensors.

To remove the process variations among the PRO instances, we calculate the Frequency Drop Ratio for each PRO instance following Equation 7.6. We first measure the Frequency Drop Ratio for all the sensors. Then, we take the average of the frequency drop of the 4 PROs in each row. The results are shown in Figure 7.15. We observe that as the PRO sensors are placed closer to the power wasters (from Row 0 to Row 8), the Frequency Drop Ratio increases. Therefore, we can see the spatial distance of the PRO sensor to the switching logic (power wasters) indeed can be reflected in the Frequency Drop Ratio. We can further use this feature to detect the location of injected faults on the chip (will be demonstrated in section 7.6). Note that there is an outlier in our designed sensor, which might be attributed to the power distribution network structure of the electronic circuits in which the power in the center of the chip is built to be more stable [121].

7.6 Fault Detection

In this section, we focus on evaluating our proposed on-die PRO sensor's performance in sensing the occurrence of fault injection attacks. We show that PRO can be used to protect the circuit from adversaries who have physical access or remote control of the device [96] which enables them to inject power or EM faults. However, we assume that PRO itself is protected against the manipulation of the attacker. We demonstrate that PRO can not only detect the occurrence of a power-based fault, but also the sensor array can detect the location of the power fault. This enables the designer (or the system administrator) to identify the source of the fault injection or the malicious circuits and build highly targeted fault response mechanisms accordingly. Moreover, We further demonstrate that PRO can be used for EM

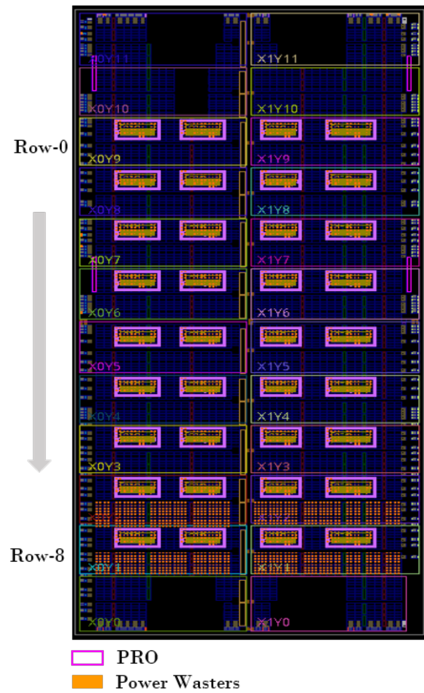

Figure 7.14: FPGA Floorplan for Evaluating PRO Performance with Regard of Sensor Locality

fault detection as well.

7.6.1 Power Fault Detection

Sharing the same PDN between a potential adversary and a victim opens the door to a new array of attacks. An adversarial logic can impose strong changes on the voltage level to cause timing faults in the victim circuit [10, 89, 96, 126]. Since all these attacks affect the PDN, we aim to build sensors that are sufficiently sensitive to the voltage level and therefore can detect such attacks. Detecting ongoing fault injection attacks will prevent resulting timing faults to go unnoticed.

Figure 7.16 shows our experimental setup for evaluating the power fault detection performance of our sensor. We instantiate AES as well as the PRO sensors array on the FPGA.

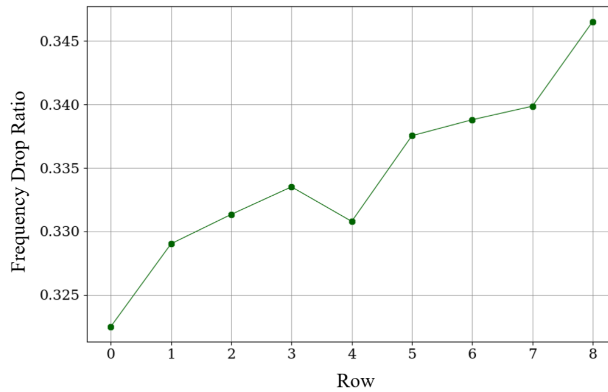

Figure 7.15: PRO's average Frequency Drop Ratio for each row versus the spatial proximity of the power wasters

Power wasters are placed locally on the chip to simulate the situation when local power faults are induced by an adversary. An on-board dip switch can control the turning on/off of the power wasters. AES control script is used to control starting the AES, send in plaintext, and read out the ciphertext. The AES control script is also used to monitor the correctness of the resulting ciphertext. We adjust the number of power wasters instantiated while AES is running. When faulty ciphertexts are observed, we know that an effective power fault is successfully injected. This ensures that the power fault detected by PRO are actually effective faults. Next, we read out the PRO's counter value through the sensor's control script both when the fault is injected and not injected respectively, and compare their values. Note that as a chip-level sensor, our goal is to detect the location of the attacker instead of identifying the fault effect within the victim algorithm/circuit.

Figure 7.17 shows the floorplan of the aforementioned setup. We placed 36 sensors on the chip and 524 power wasters are instantiated to generate power fault. We first put the power wasters at Row 1 and Row 2 on the left as shown in the orange blocks in Figure 7.17. While AES is running, we read out the sensor's counter value when the faults are injected and not injected by power wasters respectively. Then we calculate the Frequency Drop Ratio based

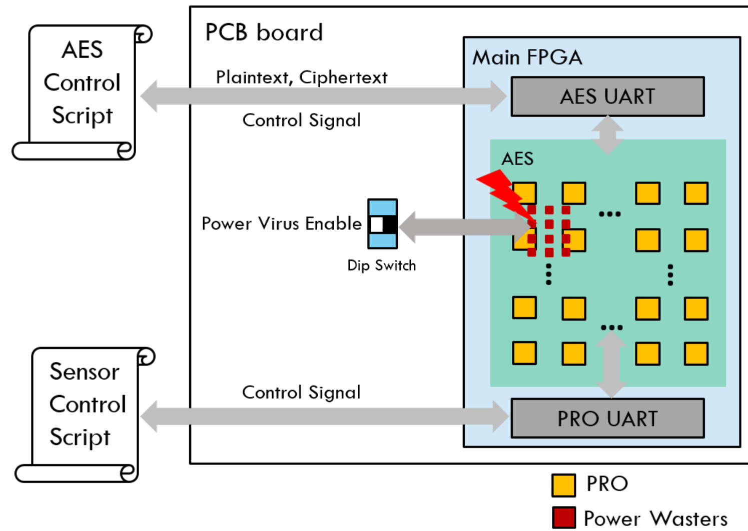

Figure 7.16: Experimental Setup for PRO Power Fault Detection

on Equation 7.6. With the PRO sensor data, we are to able find the location of the power fault. First, to locate which row has the power fault, we take the average of the 4 PRO sensors' frequency drop ratio in each row. Figure 7.18 shows the result of each row's average Frequency Drop Ratio. The maximum Frequency Drop Ratio points to a location adjacent to Row 2. This demonstrates that our sensor array can point to the correct row that the fault has occurred. Then, we divide the chip into two regions, left and right. To locate the fault region, we take the average of the Frequency Drop Ratios of the 18 sensors in the left and right two columns separately. The average Frequency Drop Ratio on the left region is 0.2184, and the average frequency drop for the right region is 0.213. The left region is higher than the right region, which indicates that the source of the fault is in the left region. This demonstrates that our sensor array can point to the correct fault column. Now, after analyzing the data of the sensor array, we can locate the power fault's location at Row 2, left region.

To further demonstrate the capability of the proposed PRO sensor in detecting the location of the fault, we placed the power wasters in different locations to inject fault while AES

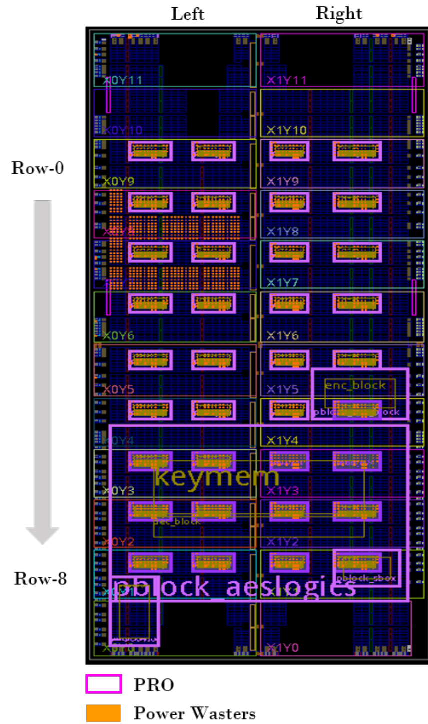

Figure 7.17: FPGA floorplan for Evaluating PRO Performance in Power Fault Detection, power wasters simulate local power fault happens at location-1.

is running. We repeat the same experimental scenario to locate the faulty row and faulty column. We first put the power wasters in the location Row 4 and Row 5 on the left region and gets the result of locating the faulty row as demonstrated in Figure 7.19. The highest frequency drop ratio points to Row 4, which indicates that the fault happens adjacent to Row 4. By analyzing the faulty column, we see the left region's average frequency drop is 0.2159 and the right region's frequency drop is 0.2091 which indicates that the left region has the fault. Therefore, the sensor array locates the place where the fault is injected is at the left region Row 4 which meets our expectation. Next, we move the power wasters to another location at Row 1 and Row 2 on the right as shown in Figure 7.20. We observe the left region's average frequency drop is 0.2083 and the right region's frequency drop is 0.2204 which indicates that the right region has the fault. As shown in the result of analyzing the faulty row in Figure 7.19, the highest frequency drop ratio points to Row 2 correctly.

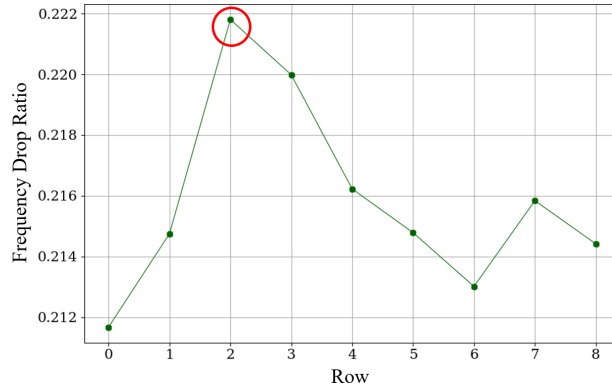

Figure 7.18: PRO average frequency drop ratio for each row when power fault happens at location-1

Therefore, we demonstrate that our proposed sensor can detect the location of the on-chip power fault.

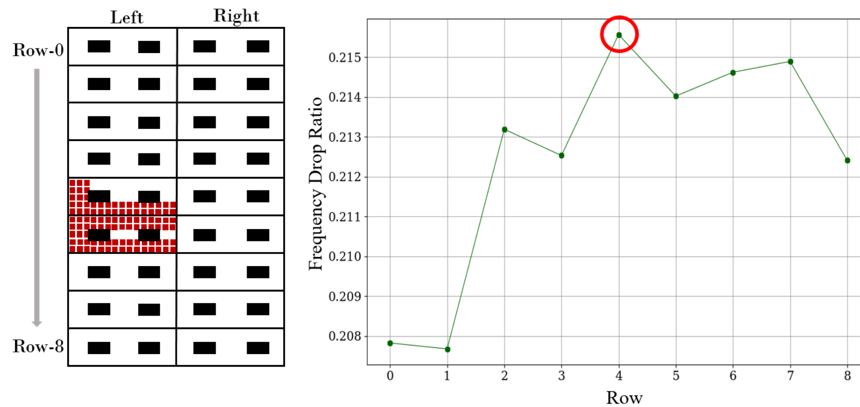

Figure 7.19: Floorplan and corresponding PRO average frequency drop ratio for each row when power fault happens at location-2. Black blocks donate PROs in the floorplan, red blocks donate power wasters positions in the floorplan.

7.6.2 Electromagnetic Fault Injection (EMFI) Detection

EMFI is a well-known active attack and describes the use of an active probe to apply an intense and transient magnetic field to Integrated Circuits (ICs). EM pulse causes a sudden current flow in the circuit of the targeted IC and therefore, the local supply voltage

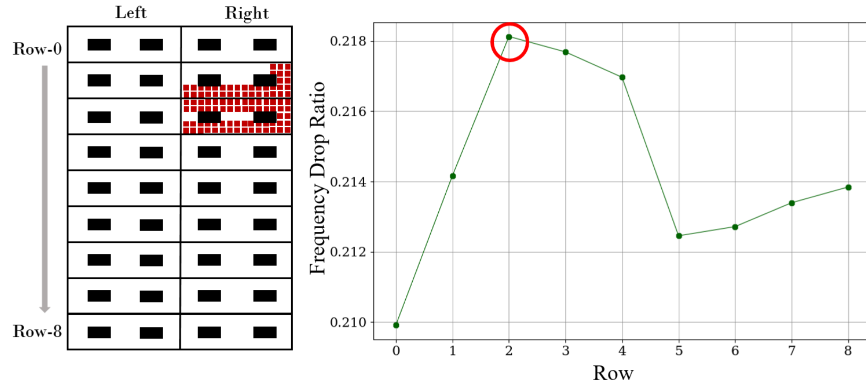

Figure 7.20: Floorplan and the corresponding PRO average frequency drop ratio for each row when power fault happens at location-3. Black blocks denote PROs in the floorplan, red blocks denote power wasters positions in the floorplan.

drops. The voltage drop reflects in the form of power consumption peaks. This produces timing faults such as bit-flips, bit-sets, and bit-resets due to timing constraint violation and sampling faults by disrupting the switching process of D-Flip Flops if EM perturbations are synchronous with clock rising edges. This enables the adversary to exploit such faults to extract sensitive content from the device. Previous research has shown that EM perturbations can cause faulty computations, alter the program flow, and cause bit-flips in the contents of the memory. Other authors have demonstrated that EM can induce faults into the devices [64, 124]. In the past few years, EM fault injection attack has gained increasing attention. In this section, we investigate the performance of our proposed PRO sensor with regard to EM fault injection.

Figure 7.21 shows the experimental setup for evaluating the EM fault injection detection performance. In this setup, we instantiate AES and the PRO array with 36 sensors on the FPGA. AES control script is used to control the starting of the AES, send in the plaintext, and read out the ciphertext. While AES is running, the EM probe is placed in a fixed position on top of the FPGA chip surface with a vertical distance of approximately 1.5mm and generates an EM pulse to induce faults. The EM probe's tip is 4mm in diameter and

produces a magnetic field that is perpendicular to the surface of the chip. A glitch controller controls the time and intensity of the EM pulse. While AES is running, we adjust the intensity of the EM pulse. When a faulty ciphertext is observed, we know that an effective EM fault is injected. Next, in each measurement, we read out the PRO sensor's counter value through the sensor's control script when the fault is injected and not injected respectively, and compare their values.

We collect 1000 frequency measurements for all 36 PROs. For each PRO sensor, we investigate the distribution of the 1000 frequency measurements when the EM fault is injected and not injected. We observe that the EM fault can cause variations of the PRO's frequency distribution. Figure 7.22 shows comparisons of the frequency distribution when the EM fault is injected and not injected for RO-0 to RO-15. We notice that the PRO sensor's frequency shifts to a larger value when faults are injected. We also observe that besides frequency shifting, there is another fault injection reaction that the PRO sensors can have. We observe that EM faults can also cause faulty counter value for RO-23 to RO-27 and RO-31 to RO-36. When the faults are injected, the counter values are read out by the UART jump to an extremely huge (and faulty) value of 4.08×10^7 MHz. Therefore, by monitoring the value of the PRO counters, we can detect ongoing Electromagnetic Fault Injection (EMFI) at run-time.

7.7 Conclusion

In this work, we proposed a multi-purpose Ring Oscillator design. We demonstrated that it is possible, with a low cost, to have a side-channel countermeasure and fault detection

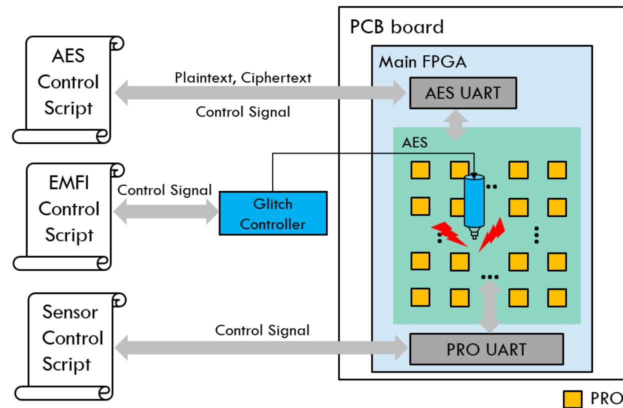

Figure 7.21: Experimental Setup for PRO EM Fault Detection

mechanism within the same design. We showed that PRO can provide an effective hiding countermeasure to the circuit with low overhead by injecting random frequency noise. We further demonstrated that the PRO array can form a comprehensive on-chip secure monitoring network. The network can potentially provide both temporal and spatial coverage of on-chip power monitoring and fault detection. PRO has the flexibility for the user to communicate and control its configurations, such as its oscillation frequency, in real-time. This feature highlights its potential to be integrated into large designs, such as SoCs, as a secure extension to build more comprehensive side-channel and fault-resistant systems. As the future work, we will further investigate integrating PRO into an SoC and build up a real-time side-channel countermeasure and fault detection/response system that can protect both software and hardware applications.

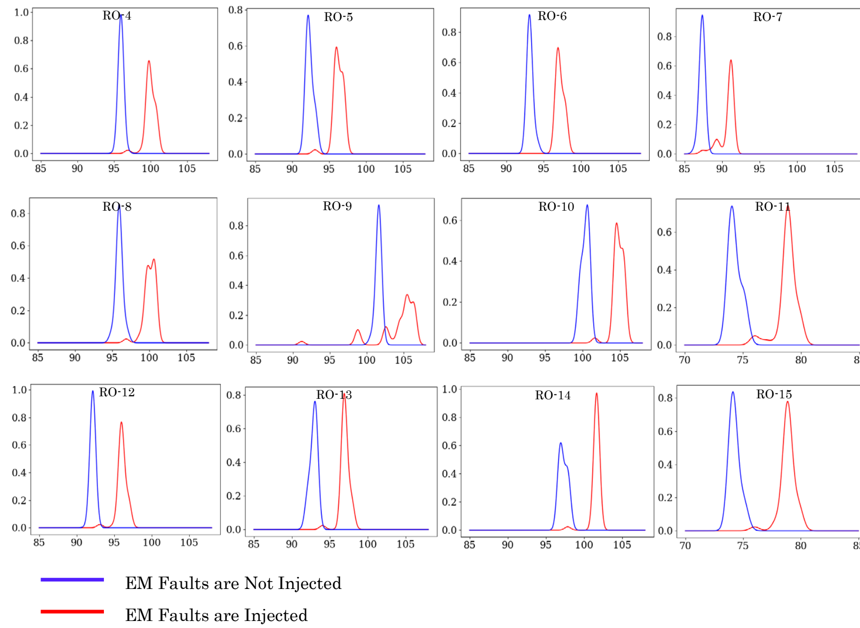

Figure 7.22: Influence on the Frequency Distribution, X-axis is probability and Y-axis is frequency.

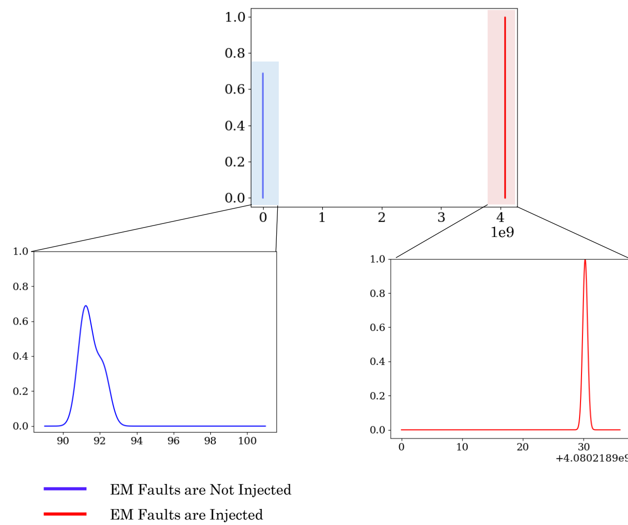

Figure 7.23: Influence on the Frequency Distribution for PRO-32

Chapter 8

Overall Conclusion

In this dissertations, we propose several advances in side-channel leakage evaluation and mitigation for the embedded system design. We first investigate the effects of integration within the embedded system, develop a novel attack model and its corresponding countermeasures. We demonstrate that even though each component of the embedded system is well-protected, new vulnerabilities may be introduced by integration. Progressively, in order to better address vulnerabilities within the embedded system, we propose novel methodologies to help designers precisely identify and mitigate the side-channel vulnerabilities at early design stages. We first looked into the simulation-based side-channel leakage assessment and evaluated each simulation abstract level's trade-offs in capturing side-channel leakage at the pre-silicon stage. Then, we develop a gate-level netlist analysis methodology- Presilicon Architecture Correlation Analysis (PACA) that enables designers to precisely identify the source of side-channel leakage in a design at the granularity of a single cell. Based on PACA, we further propose selective replacement as a low-cost side-channel countermeasure, and we show that side-channel leakage can be significantly reduced by only protecting the most leaky cells in a design. Additionally, we develop an improvement on TVLA; we show that the proposed bootstrap can significantly accelerate the leakage decreasing. In the last part of this dissertation, we developed a multipurpose primitive - PRO, which can proactively protect the design's PDN against side-channel and fault attacks. We conclude that PRO can serve as an application-independent multipurpose countermeasure to address on-chip side-channel and fault vulnerabilities at a low cost.

Bibliography

[1] FortifyIQ. https://www.fortifyiq.com/.

[2] Virtualyzr. https://cadforassurance.org/tools/design-for-trust/virtualyzr/.

[3] Botan Library. https://botan.randombit.net/.

[4] Libgcrypt Library. https://gnupg.org/software/libgcrypt/index.html.

[5] mbedTLS Library. https://tls.mbed.org/.

[6] OpenSSL Library. https://www.openssl.org/.

[7] Riscure. https://www.riscure.com/, .

[8] RISC-V. https://github.com/cliffordwolf/picorv32/, .

[9] Astronomers reevaluate the age of the universe. https://www.space.com/universe-age-14-billion-years-old.

[10] Md Mahbub Alam, Shahin Tajik, Fatemeh Ganji, Mark Tehranipoor, and Domenic Forte. Ram-jam: Remote temperature and voltage fault attack on fpgas using memory collisions. In *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 48–55, 2019. doi: 10.1109/FDTC.2019.00015.

[11] F. Amiel, C. Clavier, and M. Tunstall. Fault analysis of dpa-resistant algorithms. In *Proc. of FDTC'06*, pages 223–236, 2006. doi: 10.1007/11889700_20. URL https://doi.org/10.1007/11889700_20.

[12] Sean Eron Anderson. Bit Twiddling Hacks. https://graphics.stanford.edu/~seander/bithacks.html.

[13] Jason Andress. *The basics of information security: understanding the fundamentals of InfoSec in theory and practice*. Syngress, 2014.

[14] Md Toufiq Hasan Anik, Mohammad Ebrahimabadi, Hamed Pirsiavash, Jean-Luc Danger, Sylvain Guilley, and Naghmeh Karimi. On-chip voltage and temperature digital sensor for security, reliability, and portability. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 506–509. IEEE, 2020.

[15] Florian Bache, Christina Plump, and Tim Güneysu. Confident leakage assessment—–a side-channel evaluation framework based on confidence intervals. In *DATE 2018*, pages 1117–1122. IEEE, 2018.

[16] J. Balasch, B. Gierlichs, R. Verdult, L. Batina, and I. Verbauwhede. Power analysis of Atmel CryptoMemory – recovering secret keys from secure EEPROMS. In Orr Dunkelman, editor, *CT-RSA 2012*, volume 7178 of *LNCS*, pages 19–34. Springer, 2012.

[17] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In Marc Joye and Amir Moradi, editors, *CARDIS 2014*, volume 8968 of *LNCS*, pages 64–81. Springer, 2015.

[18] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[19] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni. Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In *Proc.*

of the WESS'10, pages 7:1–7:10, 2010. doi: 10.1145/1873548.1873555. URL http://doi.acm.org/10.1145/1873548.1873555.

[20] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. Countermeasures against fault attacks on software implemented aes: effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security*, page 7. ACM, 2010.

[21] George Becker, J Cooper, Elke DeMulder, Gilbert Goodwill, Joshua Jaffe, G Kenworthy, T Kouzminov, A Leiserson, M Marson, Pankaj Rohatgi, et al. Test vector leakage assessment (tvla) methodology in practice. In *International Cryptographic Module Conference*, volume 1001, page 13, 2013.

[22] E. Biham. A fast new DES implementation in software. In *Proc. of FSE'97*, pages 260–272, 1997. doi: 10.1007/BFb0052352. URL http://dx.doi.org/10.1007/BFb0052352.

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

[24] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. Present: An ultra-lightweight block cipher. In *International workshop on cryptographic hardware and embedded systems*, pages 450–466. Springer, 2007.

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

[26] A. Boscher and H. Handschuh. Masking does not protect against differential fault

attacks. In *Proc. of FDTC'08*, pages 35–40, 2008. doi: 10.1109/FDTC.2008.12. URL https://doi.org/10.1109/FDTC.2008.12.

[27] Jakub Breier, Dirmanto Jap, and Chien-Ning Chen. Laser profiling for the back-side fault attacks: with a practical laser skip instruction attack on aes. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*, pages 99–103. ACM, 2015.

[28] E. Brier, C. Clavier, and F. Olivier. Correlation power analysis with a leakage model. In M. Joye and J.-J. Quisquater, editors, *CHES 2004*, volume 3156 of *LNCS*, pages 16–29. Springer, 2004.

[29] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *International workshop on cryptographic hardware and embedded systems*, pages 16–29. Springer, 2004.

[30] Olivier Bronchain, Tobias Schneider, and François-Xavier Standaert. Multi-tuple leakage detection and the dependent signal issue. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, (2):318–345, 2019.

[31] Cadence Design Systems Inc. *Cadence Spectre AMS Designer, Version 18.1.0.* San Jose, CA. URL https://www.cadence.com/.

[32] Cadence Design Systems Inc. *Cadence Virtuoso, Version 6.1.7.* San Jose, CA, 2018. URL https://www.cadence.com/.

[33] Zhimin Chen, Syed Haider, and Patrick Schaumont. Side-channel leakage in masked circuits caused by higher-order circuit effects. In Jong Hyuk Park, Hsiao-Hwa Chen, Mohammed Atiquzzaman, Changhoon Lee, Tai-Hoon Kim, and Sang-Soo Yeo, editors, *Advances in Information Security and Assurance, Third International Conference and Workshops, ISA 2009, Seoul, Korea, June 25-27, 2009. Proceedings*, volume 5576 of

Lecture Notes in Computer Science, pages 327–336. Springer, 2009. doi: 10.1007/978-3-642-02617-1_34. URL https://doi.org/10.1007/978-3-642-02617-1_34.

[34] T. Cnudde, O. Reparaz, B. Bilgin, S. Nikova, V. Nikov, and V. Rijmen. Masking AES with d+1 shares in hardware. *IACR Cryptology ePrint Archive*, 2016:631, 2016. URL http://eprint.iacr.org/2016/631.

[35] Lucian Cojocar, Kostas Papagiannopoulos, and Niek Timmers. Instruction duplication: Leaky and not too fault-tolerant! In *International Conference on Smart Card Research and Advanced Applications*, pages 160–179. Springer, 2017.

[36] Jean-Sébastien Coron and Ilya Kizhvatov. Analysis and improvement of the random delay countermeasure of ches 2009. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 95–109. Springer, 2010.

[37] Yann Le Corre, Johann Großschädl, and Daniel Dinu. Micro-architectural power simulator for leakage assessment of cryptographic software on ARM cortex-m3 processors. In Junfeng Fan and Benedikt Gierlichs, editors, *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*, volume 10815 of *Lecture Notes in Computer Science*, pages 82–98. Springer, 2018. doi: 10.1007/978-3-319-89641-0_5. URL https://doi.org/10.1007/978-3-319-89641-0_5.

[38] Haotian Dai and Selçuk Köse. On the vulnerability of hardware masking in practical implementations. In *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, pages 77–82, 2021.

[39] Debayan Das, Shovan Maity, Saad Bin Nasir, Santosh Ghosh, Arijit Raychowdhury, and Shreyas Sen. High efficiency power side-channel attack immunity using noise

injection in attenuated signature domain. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 62–67. IEEE, 2017.

[40] Nicolas Debande, Maël Berthier, Yves Bocktaels, and Thanh-Ha Le. Profiled model based power simulator for side channel evaluation. *IACR Cryptology ePrint Archive*, 2012:703, 2012. URL http://eprint.iacr.org/2012/703.

[41] E.Berker and J.Kelsey. Recommendation for random number generation using deterministic random bit generators.

[42] B Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1):1–26, 1979.

[43] T. Eisenbarth, T. Kasper, A. Moradi, C. Paar, M. Salmasizadeh, and M.T. Manzuri Shalmani. On the power of power analysis in the real world: A complete break of the KeeLoq code hopping scheme. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 203–220. Springer, 2008.

[44] S. Endo, N. Homma, Y. Hayashi, J. Takahashi, H. Fuji, and T. Aoki. A multiple-fault injection attack by adaptive timing control under black-box conditions and a countermeasure. In *Proc. of COSADE'14*, pages 214–228, 2014.

[45] Blömer et al. Provably secure masking of aes. In *International workshop on selected areas in cryptography*, pages 69–83. Springer, 2004.

[46] Chen et al. Dual-rail random switching logic: a countermeasure to reduce side channel leakage. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 242–254. Springer, 2006.

[47] De Cnudde et al. Does coupling affect the security of masked implementations? In

International Workshop on Constructive Side-Channel Analysis and Secure Design, pages 1–18. Springer, 2017.

[48] Dyrkolbotn et al. Security implications of crosstalk in switching cmos gates. In *International Conference on Information Security*, pages 269–275. Springer, 2010.

[49] Guilley et al. Differential power analysis model and some results. In *Smart Card Research and Advanced Applications Vi*, pages 127–142. Springer, 2004.

[50] Leiserson et al. Gate-level masking under a path-based leakage metric. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 580–597. Springer, 2014.

[51] Moradi et al. Correlation-enhanced power analysis collision attack. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 125–139. Springer, 2010.

[52] Mulder et al. Identifying and eliminating side-channel leaks in programmable systems. *IEEE Des. Test*, 35(1):74–89, 2018. doi: 10.1109/MDAT.2017.2766166. URL https://doi.org/10.1109/MDAT.2017.2766166.

[53] Oswald et al. A side-channel analysis resistant description of the aes s-box. In *International Workshop on Fast Software Encryption*, pages 413–423. Springer, 2005.

[54] Standaert et al. Towards security limits in side-channel attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 30–45. Springer, 2006.

[55] Suzuki et al. Security evaluation of dpa countermeasures using dual-rail pre-charge logic style. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 255–269. Springer, 2006.

[56] Trichina et al. Small size, low power, side channel-immune aes coprocessor: design and synthesis results. In *International Conference on Advanced Encryption Standard*, pages 113–127. Springer, 2004.

[57] Wang et al. Role of power grid in side channel attack and power-grid-aware secure design. In *Proceedings of the 50th Annual Design Automation Conference*, pages 1–9, 2013.

[58] Yao et al. Augmenting leakage detection using bootstrapping.

[59] Yao et al. Architecture correlation analysis: Identifying the source of side-channel leakage at gate-level. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST 2020)*. IEEE, 2020.

[60] Thomas Fuhr, Eliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault attacks on aes with faulty ciphertexts only. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 108–118. IEEE, 2013.

[61] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In C. K. Koç, D. Naccache, and C. Paar, editors, *CHES 2001*, volume 2162 of *LNCS*, pages 251–261. Springer, 2001.

[62] Vinod Ganesan, Rahul Bodduna, Chester Rebeiro, et al. Param: A microprocessor hardened for power side-channel attack resistance. *arXiv preprint arXiv:1911.08813*, 2019.

[63] Daniel Genkin, Itamar Pipman, and Eran Tromer. Get your hands off my laptop: Physical side-channel key-extraction attacks on pcs. In Lejla Batina and Matthew Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 242–260. Springer, 2014.

[64] Marjan Ghodrati, Bilgiday Yuce, Surabhi Gujar, Chinmay Deshpande, Leyla Nazhandali, and Patrick Schaumont. Inducing local timing fault through em injection. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018. doi: 10.1109/DAC.2018.8465836.

[65] B Gierlichs, L Batina, P Tuyls, and B Preneel. Mutual information analysis. a generic side-channel distinguisher. ches'08, lncs, 2008.

[66] Dennis R.E. Gnad, Fabian Oboril, Saman Kiamehr, and Mehdi B. Tahoori. Analysis of transient voltage fluctuations in fpgas. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 12–19, 2016. doi: 10.1109/FPT.2016.7929182.

[67] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

[68] Andreas Gornik, Amir Moradi, Jurgen Oehm, and Christof Paar. A Hardware-Based Countermeasure to Reduce Side-Channel Leakage: Design, Implementation, and Evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1308–1319, August 2015. ISSN 0278-0070, 1937-4151. doi: 10.1109/TCAD.2015.2423274. URL http://ieeexplore.ieee.org/document/7087376/.

[69] Joseph Gravellier, Jean-Max Dutertre, Yannick Teglia, and Philippe Loubet-Moundi. High-speed ring oscillator based sensors for remote side-channel attacks on fpgas. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2019. doi: 10.1109/ReConFig48160.2019.8994789.

[70] H. Groß, S. Mangard, and T. Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. Cryptology ePrint Archive, Report 2016/486, 2016. http://eprint.iacr.org/.

[71] Xiaofei Guo and Ramesh Karri. Recomputing with permuted operands: A concurrent error detection approach. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(10):1595–1608, 2013.

[72] Miao Tony He, Jungmin Park, Adib Nahiyan, Apostol Vassilev, Yier Jin, and Mark Tehranipoor. RTL-PSC: automated power side-channel leakage assessment at register-transfer level. In *37th IEEE VLSI Test Symposium, VTS 2019, Monterey, CA, USA, April 23-25, 2019*, pages 1–6, 2019. doi: 10.1109/VTS.2019.8758600. URL https://doi.org/10.1109/VTS.2019.8758600.

[73] Wei He, Jakub Breier, Shivam Bhasin, Noriyuki Miura, and Makoto Nagata. Ring oscillator under laser: potential of pll-based countermeasure against laser fault injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 102–113. IEEE, 2016.

[74] ARM Holdings. Procedure call standard for the arm architecture, 2013.

[75] HM James Hung, Robert T O'Neill, Peter Bauer, and Karl Kohne. The behavior of the p-value when the alternative hypothesis is true. *Biometrics*, pages 11–22, 1997.

[76] SPARC International Inc and David L Weaver. *The SPARC architecture manual*. Prentice-Hall, 1994.

[77] Y. Ishai, A. Sahai, and D. Wagner. Private circuits: Securing hardware against probing attacks. In *Proc. of CRYPTO'03*, pages 463–481. Springer, 2003.

[78] Ramesh Karri, Kaijie Wu, Piyush Mishra, and Yongkook Kim. Concurrent error detection of fault-based side-channel cryptanalysis of 128-bit symmetric block ciphers. In *Proceedings of the 38th annual Design Automation Conference*, pages 579–584. ACM, 2001.

[79] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In *Proc. of CHES'09*, pages 1–17, 2009. doi: 10.1007/978-3-642-04138-9_1. URL http://dx.doi.org/10.1007/978-3-642-04138-9_1.

[80] M. Kasper, T. Kasper, A. Moradi, and C. Paar. Breaking KeeLoq in a flash: On extracting keys at lightning speed. In Bart Preneel, editor, *AFRICACRYPT*, volume 5580 of *LNCS*, pages 403–420. Springer, 2009.

[81] B. Kelsey, J.and Schneier, D. Wagner, and C. Hall. Cryptanalytic attacks on pseudo-random number generators. In *Proc. of FSE'98*, pages 168–188. Springer, 1998.

[82] Chan-Kyung Kim, Bai-Sun Kong, Chil-Gee Lee, and Young-Hyun Jun. Cmos temperature sensor with ring oscillator for mobile dram self-refresh control. In *2008 IEEE International Symposium on Circuits and Systems*, pages 3094–3097. IEEE, 2008.

[83] Chong Hee Kim and Jean-Jacques Quisquater. Fault attacks for crt based rsa: New attacks, new results, and new countermeasures. In *IFIP International Workshop on Information Security Theory and Practices*, pages 215–228. Springer, 2007.

[84] Chong Hee Kim and Jean-Jacques Quisquater. Faults, injection methods, and fault attacks. *IEEE Design & Test of Computers*, 24(6):544–545, 2007.

[85] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *CRYPTO '96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[86] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In M. J. Wiener, editor, *CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.

[87] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[88] Paul Kocher, Ruby Lee, Gary McGraw, and Anand Raghunathan. Security as a new dimension in embedded system design. In *Proceedings of the 41st annual Design Automation Conference*, pages 753–760, 2004.

[89] Jonas Krautter, Dennis RE Gnad, and Mehdi B Tahoori. Fpgahammer: Remote voltage fault attacks on shared fpgas, suitable for dfa on aes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 44–68, 2018.

[90] Jonas Krautter, Dennis R.E. Gnad, Falk Schellenberg, Amir Moradi, and Mehdi B. Tahoori. Active fences against voltage-based side channels in multi-tenant fpgas. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019. doi: 10.1109/ICCAD45719.2019.8942094.

[91] Jonas Krautter, Dennis Gnad, and Mehdi Tahoori. Cpamap: On the complexity of secure fpga virtualization, multi-tenancy, and physical design. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 121–146, 2020.

[92] J. Lalande, K. Heydemann, and P. Berthomé. Software countermeasures for control flow integrity of smart card c codes. In *Proc. of ESORICS'14*, pages 200–218. Springer, 2014.

[93] Y. Li, K. Sakiyama, S. Gomisawa, T. Fukunaga, J. Takahashi, and K. Ohta. Fault sensitivity analysis. In *CHES*, volume 6225, pages 320–334. Springer, 2010.

[94] Po-Chun Liu, Hsie-Chia Chang, and Chen-Yi Lee. A low overhead dpa countermeasure circuit based on ring oscillators. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 57(7):546–550, 2010. doi: 10.1109/TCSII.2010.2048400.

[95] Po-Chun Liu, Hsie-Chia Chang, and Chen-Yi Lee. A low overhead dpa countermeasure

circuit based on ring oscillators. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 57(7):546–550, 2010.

[96] Dina Mahmoud and Mirjana Stojilović. Timing violation induced faults in multi-tenant fpgas. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1745–1750, 2019. doi: 10.23919/DATE.2019.8715263.

[97] Paolo Maistri, Pierre Vanhauwaert, and Régis Leveugle. A novel double-data-rate aes architecture resistant against fault injection. In *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on*, pages 54–61. IEEE, 2007.

[98] S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks: Revealing the secrets of smart cards*. Springer Publishing Company, Incorporated, 2010.

[99] Stefan Mangard and Kai Schramm. Pinpointing the side-channel leakage of masked AES hardware implementations. In *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, pages 76–90, 2006. doi: 10.1007/11894063_7. URL https://doi.org/10.1007/11894063_7.

[100] Stefan Mangard, Thomas Popp, and Berndt M Gammel. Side-channel leakage of masked cmos gates. In *Cryptographers' Track at the RSA Conference*, pages 351–365. Springer, 2005.

[101] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[102] Stefan Mangard, Elisabeth Oswald, and F-X Standaert. One for all–all for one: uni-

fying standard differential power analysis attacks. *IET Information Security*, 5(2): 100–110, 2011.

[103] David McCann, Carolyn Whitnall, and Elisabeth Oswald. ELMO: emulating leaks for the ARM cortex-m0 without access to a side channel lab. *IACR Cryptol. ePrint Arch.*, 2016:517, 2016. URL http://eprint.iacr.org/2016/517.

[104] Mark S Merkow and Jim Breithaupt. *Information security: Principles and practices*. Pearson Education, 2014.

[105] Noriyuki Miura, Zakaria Najm, Wei He, Shivam Bhasin, Xuan Thuy Ngo, Makoto Nagata, and Jean-Luc Danger. Pll to the rescue: A novel em fault countermeasure. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016. doi: 10.1145/2897937.2898065.

[106] Shayan Moini, Xiang Li, Peter Stanwicks, George Provelengios, Wayne Burleson, Russell Tessier, and Daniel Holcomb. Understanding and comparing the capabilities of on-chip voltage sensors against remote power attacks on fpgas. In *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 941–944. IEEE, 2020.

[107] A. Moradi, A. Barenghi, T. Kasper, and C. Paar. On the vulnerability of FPGA bitstream encryption against power analysis attacks: extracting keys from Xilinx Virtex-II FPGAs. In *CCS*, pages 111–124, 2011.

[108] Amir Moradi. Side-channel leakage through static power - should we care about in practice? In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer*

Science, pages 562–579. Springer, 2014. doi: 10.1007/978-3-662-44709-3_31. URL https://doi.org/10.1007/978-3-662-44709-3_31.

[109] Amir Moradi. Side-channel leakage through static power. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 562–579. Springer, 2014.

[110] Amir Moradi, Bastian Richter, Tobias Schneider, and François-Xavier Standaert. Leakage detection with the χ^2-test. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, (1):209–237, 2018.

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

[112] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. *arXiv preprint arXiv:1402.6421*, 2014.

[113] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4(3):145–156, 2014.

[114] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *International conference on information and communications security*, pages 529–545. Springer, 2006.

[115] Jason Oberg, Sarah Meiklejohn, Timothy Sherwood, and Ryan Kastner. Leveraging gate-level properties to identify hardware timing channels. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 33(9):1288–1301, 2014. doi: 10.1109/TCAD.2014.2331332. URL https://doi.org/10.1109/TCAD.2014.2331332.

[116] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[117] S. Patranabis, A. Chakraborty, and D. Mukhopadhyay. Fault tolerant infective countermeasure for AES. In *Proc. of SPACE'15*, pages 190–209, 2015. doi: 10.1007/978-3-319-24126-5_12. URL http://dx.doi.org/10.1007/978-3-319-24126-5_12.

[118] Nicholas D. Pattengale, Masoud Alipour, Olaf R. P. Bininda-Emonds, Bernard M. E. Moret, and Alexandros Stamatakis. How many bootstrap replicates are necessary? *Journal of Computational Biology*, 17(3):337–354, 2010.

[119] Philippe Pierre Pebay. Formulas for robust, one-pass parallel computation of covariances and arbitrary-order statistical moments. Technical report, Sandia National Laboratories, 2008.

[120] Eric Peeters, François-Xavier Standaert, and Jean-Jacques Quisquater. Power and electromagnetic analysis: Improved model, consequences and comparisons. *Integration, the VLSI journal*, 40(1):52–60, 2007.

[121] Mikhail Popovich, Andrey Mezhiba, and Eby G Friedman. *Power distribution networks with on-chip decoupling capacitors*. Springer Science & Business Media, 2007.

[122] Thomas Popp and Stefan Mangard. Masked dual-rail pre-charge logic: Dpa-resistance without routing constraints. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 172–186. Springer, 2005.

[123] Thomas Popp, Mario Kirschbaum, Thomas Zefferer, and Stefan Mangard. Evaluation of the masked logic style mdpl on a prototype chip. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 81–94. Springer, 2007.

[124] F. Poucheret, K. Tobich, M. Lisarty, L. Chusseauz, B. Robissonx, and P. Maurine. Local and direct em injection of power into cmos integrated circuits. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 100–104, 2011. doi: 10.1109/FDTC.2011.18.

[125] George Provelengios, Daniel Holcomb, and Russell Tessier. Characterizing power distribution attacks in multi-user fpga environments. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 194–201. IEEE, 2019.

[126] George Provelengios, Daniel Holcomb, and Russell Tessier. Power distribution attacks in multitenant fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(12):2685–2698, 2020. doi: 10.1109/TVLSI.2020.3027711.

[127] J.-J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In I. Attali and T. P. Jensen, editors, *E-smart 2001*, volume 2140 of *LNCS*, pages 200–210. Springer, 2001.

[128] Francesco Regazzoni, Stéphane Badel, Thomas Eisenbarth, Johann Großschädl, Axel Poschmann, Zeynep Toprak Deniz, Marco Macchetti, Laura Pozzi, Christof Paar, Yusuf Leblebici, and Paolo Ienne. A simulation-based methodology for evaluating the dpa-resistance of cryptographic functional units with application to CMOS and MCML technologies. In *Proceedings of the 2007 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2007), Samos, Greece, July 16-19, 2007*, pages 209–214, 2007. doi: 10.1109/ICSAMOS.2007.4285753. URL https://doi.org/10.1109/ICSAMOS.2007.4285753.

[129] Oscar Reparaz, Benedikt Gierlichs, and Ingrid Verbauwhede. Fast leakage assessment. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 387–399. Springer, 2017.

[130] M. Rivain and E. Prouff. Provably secure higher-order masking of AES. In *Proc. of CHES'10, 2010*, pages 413–427, 2010. doi: 10.1007/978-3-642-15031-9_28. URL https://doi.org/10.1007/978-3-642-15031-9_28.

[131] T. Roche, V. Lomné, and K. Khalfallah. Combined fault and side-channel attack on protected implementations of AES. In *Proc. of CARDIS'11*, pages 65–83, 2011. doi: 10.1007/978-3-642-27257-8_5. URL https://doi.org/10.1007/978-3-642-27257-8_5.

[132] N. Savić, M. Stojcev, and T. Nikolić. Fault tolerant pseudorandom number generator. In *Proc. of MECO'12*, pages 30–33, June 2012.

[133] Jörn-Marc Schmidt and Christoph Herbst. A practical fault attack on square and multiply. In *Fault Diagnosis and Tolerance in Cryptography, 2008. FDTC'08. 5th Workshop on*, pages 53–58. IEEE, 2008.

[134] Tobias Schneider and Amir Moradi. Leakage assessment methodology. In Tim Güneysu and Helena Handschuh:, editors, *CHES 2015*, volume 9293 of *LNCS*, pages 495–513. Springer, 2015.

[135] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. *CoRR*, abs/1912.05183, 2019. URL http://arxiv.org/abs/1912.05183.

[136] Danilo Sijacic, Josep Balasch, Bohan Yang, Santosh Ghosh, and Ingrid Verbauwhede. Towards efficient and automated side channel evaluations at design time. In *PROOFS 2018, 7th International Workshop on Security Proofs for Embedded Systems, colocated with CHES 2018, Amsterdam, The Netherlands, September 13, 2018*, pages 16–31, 2018. URL http://www.easychair.org/publications/paper/xPnF.

[137] Patanjali SLPSK, Prasanna Karthik Vairam, Chester Rebeiro, and Kamakoti Veezhinathan. Karna: A gate-sizing based security aware eda flow for improved power side-channel attack protection. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 04-07, 2019*.

[138] François-Xavier Standaert. How (not) to use Welch's t-test in side-channel security evaluations. In Begül Bilgin and Jean-Bernard Fischer, editors, *CARDIS 2018*, volume 11389 of *LNCS*, pages 65–79. Springer, 2019.

[139] O-X Standaert, Eric Peeters, Gaël Rouvroy, and J-J Quisquater. An overview of power analysis attacks against field programmable gate arrays. *Proceedings of the IEEE*, 94 (2):383–394, 2006.

[140] Shahin Tajik, Julian Fietkau, Heiko Lohrke, Jean-Pierre Seifert, and Christian Boit. Pufmon: Security monitoring of fpgas using physically unclonable functions. In *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 186–191. IEEE, 2017.

[141] Stefan Tillich and Christoph Herbst. Attacking state-of-the-art software countermeasures—a case study for aes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 228–243. Springer, 2008.

[142] Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling pc on arm using fault injection. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2016 Workshop on*, pages 25–35. IEEE, 2016.

[143] Kris Tiri and Ingrid Verbauwhede. A logic level design methodology for a secure dpa resistant asic or fpga implementation. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 1, pages 246–251. IEEE, 2004.

[144] E. Trichina. Combinational logic design for AES subbyte transformation on masked data. *IACR Cryptology ePrint Archive*, 2003:236, 2003. URL http://eprint.iacr.org/2003/236.

[145] H. Uno, S. Endo, Y. Hayashi, N. Homma, and T. Aoki. Chosen-message electromagnetic analysis against cryptographic software on embedded OS. In *EMC '14*, pages 314–317. IEEE, 2014.

[146] Dmitry Utyamishev and Inna Partin-Vaisband. Real-time detection of power analysis attacks by machine learning of power supply variations on-chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(1):45–55, 2018.

[147] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. The risc-v instruction set manual. volume 1: User-level isa, version 2.0. Technical report, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 2014.

[148] BP Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[149] Carolyn Whitnall and Elisabeth Oswald. A cautionary note regarding the usage of leakage detection tests in security evaluation. *IACR Cryptology ePrint Archive*, 2019:703, 2019. URL https://eprint.iacr.org/2019/703.

[150] M. Witteman and M. Oostdijk. Secure application programming in the presence of side channel attacks. In *RSA conference*, volume 2008, 2008.

[151] Lennart Wouters, Jan Van den Herrewegen, Flavio D Garcia, David Oswald, Benedikt Gierlichs, and Bart Preneel. Dismantling dst80-based immobiliser systems. *IACR*

Transactions on Cryptographic Hardware and Embedded Systems, 2020(2):99–127, 2019.

[152] Yuan Yao and Patrick Schaumont. A low-cost function call protection mechanism against instruction skip fault attacks. In *Proceedings of the 2018 Workshop on Attacks and Solutions in Hardware Security*, pages 55–64, 2018.

[153] Yuan Yao, Mo Yang, Conor Patrick, Bilgiday Yuce, and Patrick Schaumont. Fault-assisted side-channel analysis of masked implementations. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 57–64. IEEE, 2018.

[154] Yuan Yao, Tarun Kathuria, Baris Ege, and Patrick Schaumont. Architecture correlation analysis (aca): identifying the source of side-channel leakage at gate-level. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 188–196. IEEE, 2020.

[155] Yuan Yao, Patrick Schaumont, Jasper Van Woudenberg, Cees-Bart Breunesse, Edgar Mateos Santillan, and Steve Stecyk. Verification of power-based side-channel leakage through simulation. In *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1112–1115. IEEE, 2020.

[156] Yuan Yao, Michael Tunstall, Elke De Mulder, Anton Kochepasov, and Patrick Schaumont. Augmenting leakage detection using bootstrapping. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 104–119. Springer, 2020.

[157] Yuan Yao, Tuna Tufan, Tarun Kathuria, Baris Ege, Ulkuhan Guler, and Patrick Schaumont. Pre-silicon architecture correlation analysis (paca): Identifying and mitigating

the source of side-channel leakage at gate-level. *IACR Cryptol. ePrint Arch.*, 2021:530, 2021.

[158] Bilgiday Yuce, Chinmay Deshpande, Marjan Ghodrati, Abhishek Bendre, Leyla Nazhandali, and Patrick Schaumont. A secure exception mode for fault-attack-resistant processing. *IEEE Trans. Dependable Secur. Comput.*, 16(3):388–401, 2019. doi: 10.1109/TDSC.2018.2823767. URL https://doi.org/10.1109/TDSC.2018.2823767.

[159] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. *SIGPLAN Not.*, 47(6):99–110, June 2012. ISSN 0362-1340. doi: 10.1145/2345156.2254078. URL http://doi.acm.org/10.1145/2345156.2254078.

[160] Liwei Zhang, A Adam Ding, François Durvaux, François-Xavier Standaert, and Yunsi Fei. Towards sound and optimal leakage detection procedure. *IACR Cryptology ePrint Archive*, 2017:287, 2017.

[161] Xuehui Zhang and Mohammad Tehranipoor. Ron: An on-chip ring oscillator network for hardware trojan detection. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[162] Mark Zhao and G Edward Suh. Fpga-based remote power side-channel attacks. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 229–244. IEEE, 2018.

[163] Kenneth M Zick and John P Hayes. Low-cost sensing with ring oscillator arrays for healthier reconfigurable systems. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 5(1):1–26, 2012.