University of Massachusetts Amherst
ScholarWorks@UMass Amherst

Doctoral Dissertations | Dissertations and Theses

October 2019

Stealthy parametric hardware Trojans in VLSI Circuits

Samaneh Ghandali

Follow this and additional works at: https://scholarworks.umass.edu/dissertations_2

Part of the Hardware Systems Commons

Recommended Citation

Ghandali, Samaneh, "Stealthy parametric hardware Trojans in VLSI Circuits" (2019). *Doctoral Dissertations*. 1730.
https://scholarworks.umass.edu/dissertations_2/1730

This Open Access Dissertation is brought to you for free and open access by the Dissertations and Theses at ScholarWorks@UMass Amherst. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

STEALTHY PARAMETRIC HARDWARE TROJANS IN VLSI CIRCUITS

A Dissertation Presented

by

SAMANEH GHANDALI

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2019

Electrical and Computer Engineering

© Copyright by Samaneh Ghandali 2019

All Rights Reserved

STEALTHY PARAMETRIC HARDWARE TROJANS IN VLSI CIRCUITS

A Dissertation Presented

by

SAMANEH GHANDALI

Approved as to style and content by:

Christof Paar, Chair

Wayne Burleson, Member

Daniel Holcomb, Member

Yeon Sik Noh, Member

Robert W. Jackson, Department Chair
Electrical and Computer Engineering

ACKNOWLEDGMENTS

I am deeply grateful to my Ph.D. advisor, Prof. Christof Paar for his outstanding support and mentorship throughout my Ph.D. studies. I especially want to thank him for his encouragement and his oversight of "hot topics". I would also like to thank Prof. Daniel Holcomb, who really treats me like my second advisor, for all his support and guidance during my Ph.D. studies. Many thanks also to my Ph.D. committee members Prof. Wayn Burleson and Prof. Yeonsik Noh for taking the time and giving me valuable feedback on my work.

During my Ph.D. studies, I have shared all the ups and downs of this journey with my colleague, best friend, and my husband, Mohammad. I am truly grateful for his continuous support and kindness throughout all these years.

This thesis would not have been possible if I would not have had help in my research. I would like to thank all of you who have helped in my research, especially my co-authors: Amir Moradi, Thorben Moos, Georg T. Becker, and Maik Ender.

I would like to thank my inspiring parents, for their great support throughout my educational career and life in general. I can count myself lucky having such supportive and caring parents. Many thanks also to my sisters for their love and always being there for me.

ABSTRACT

STEALTHY PARAMETRIC HARDWARE TROJANS IN VLSI CIRCUITS

SEPTEMBER 2019

SAMANEH GHANDALI

B.Sc., SHAHED UNIVERSITY

M.Sc., SHAHID BEHESHTI UNIVERSITY

PhD, UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Christof Paar

Over the last decade, hardware Trojans have gained increasing attention in academia, industry and by government agencies. In order to design reliable countermeasures, it is crucial to understand how hardware Trojans can be built in practice. This is an area that has received relatively scant treatment in the literature. In this thesis, we examine how particularly stealthy parametric Trojans can be introduced to VLSI circuits. Parametric Trojans do not require any additional logic and are purely based on subtle manipulations on the sub-transistor level to modify the parameters of few transistors which makes them very hard to detect.

We introduce a design methodology to insert stealthy parametric hardware Trojans which are based on injecting extremely rare path delay faults into the netlist of the target circuit. As a case study, we apply our method to a 32-bit multiplier circuit resulting in a stealthy Trojan multiplier that computes faulty outputs for specific combinations of input pairs that are applied to the circuit. The multiplier can be

used to realize bug attacks, introduced by Biham et al. in 2008. We also extend this concept and show how it can be used to attack ECDH key agreement protocols. Our method is a versatile tool for designing stealthy Trojans for a given circuit and is not restricted to multipliers and the bug attack.

In this thesis we also examine how a stealthy side-channel hardware Trojan can be inserted in a provably-secure side-channel analysis protected implementation. Once the Trojan is triggered, the malicious design exhibits exploitable side-channel leakage leading to successful key recovery attacks. The underlying concept is based on a secure masked hardware implementation which does not exhibit any detectable leakage. However, by running the device at a particular clock frequency one of the requirements of the underlying masking scheme is not fulfilled anymore, and the device's side-channel leakage can be exploited. We apply our technique to a Threshold Implementation of the PRESENT block cipher realized in both FPGA and ASIC. We show that triggering the Trojan makes both FPGA and ASIC prototypes vulnerable to certain SCA attacks.

True random number generators (TRNGs) are an essential component of cryptographic designs, which are used to generate private keys for encryption and authentication, and are used in masking countermeasures. This thesis also presents a mechanism to design a stealthy parametric hardware Trojan for ring oscillator-based TRNGs. When the Trojan is triggered by operation at a specific high temperature the malicious TRNG generates predictable non-random outputs, yet under normal operating conditions it works correctly. Also we elaborate a stochastic model based on Markov Chains by which the attacker can use their knowledge of the Trojan to predict the TRNG outputs.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS . . . . . **iv**

ABSTRACT . . . . . **v**

LIST OF TABLES . . . . . **x**

LIST OF FIGURES . . . . . **xii**

CHAPTER

1. MOTIVATION **1**

1.1 Introduction 1
1.2 Organization and Contribution of the Thesis 5

2. BACKGROUND AND RELATED WORK **8**

2.1 Hardware Trojans 8
2.2 Side-Channel Based Trojans 10
2.3 Threshold Implementation 11

2.3.1 Uniformity 12

3. A DESIGN METHODOLOGY FOR STEALTHY PARAMETRIC HARDWARE TROJANS **14**

3.1 Overview 15
3.2 Delay Insertion 16

3.2.1 Decrease Width 17
3.2.2 Raise Threshold 17
3.2.3 Increase Gate Length 18

3.3 Phase I: Rare Path Selection 18

3.3.1 Controllability and Observability 19

3.3.2 Timing Graph . . . 19
3.3.3 Selecting a Path Through Timing Graph . . . 20

3.4 Phase II: Delay Distribution . . . 23

3.4.1 Constraint on Total Path Delay . . . 24
3.4.2 Constraint on Delay of Each Gate . . . 25
3.4.3 Fitness Function . . . 25

3.5 Experimental Results . . . 26

3.5.1 Evaluation of Phase I (Path Selection) . . . 26
3.5.2 Evaluation of Phase II (Delay Distribution) . . . 26
3.5.3 Overall Evaluation . . . 28

3.6 Bug Attack On ECDH with a Trojan Multiplier . . . 29

3.6.1 Fault Model of the Trojan Multiplier . . . 30
3.6.2 Case Study: An ECDH implementation with Montgomery Ladder . . . 32
3.6.3 Montgomery Ladder . . . 35

3.6.3.1 Computing the failure probability of a scalar multiplication . . . 36

4. SIDE-CHANNEL HARDWARE TROJAN FOR PROVABLY-SECURE SCA PROTECTED IMPLEMENTATIONS . . . 38

4.1 Technique . . . 39
4.2 Application . . . 44

4.2.1 Uniform TI of $\mathcal{F}$ . . . 46
4.2.2 Inserting the Trojan . . . 46

4.2.2.1 ASIC Platforms . . . 47
4.2.2.2 FPGA Platforms . . . 49

4.3 ASIC Implementation . . . 51

4.3.1 Rare Path Selection Phase . . . 51
4.3.2 Delay Distribution Phase . . . 54

4.4 FPGA Practical Results . . . 55

4.4.1 Design Architecture . . . 55
4.4.2 Noise Generator . . . 57
4.4.3 SCA Evaluations . . . 59

4.4.3.1 Measurement Setup. 59
4.4.3.2 Methodology. 59

4.5 ASIC Practical Results 63

4.5.1 Measurement Setup 68
4.5.2 SCA Results 69

4.5.2.1 Results on 90 nm ASIC 69
4.5.2.2 Results on 65 nm ASIC 72

5. TEMPERATURE-BASED HARDWARE TROJAN FOR RING-OSCILLATOR-BASED TRNGS 77

5.1 Introduction 77
5.2 Ring oscillator-based TRNG 78
5.3 Hardware Trojan RO-based TRNG 80

5.3.1 Temperature Dependence of Propagation Delay 81
5.3.2 Injecting temperature-triggered Trojan into RO-based TRNG 82

5.4 How to Predict the Output of the Trojan TRNG 83

5.4.1 Markov chain 84
5.4.2 Predicting Bits 86

5.5 Practical Results 90

5.5.1 Randomness and Performance of the TRNG 90

6. CONCLUSION 94

BIBLIOGRAPHY 97

LIST OF TABLES

Table **Page**

3.1 Computation of $diffj$ for different gate types. In the case of 2-input gates, we assume without loss of generality that input A is the on-path input and B is the off-path input. The first two columns show the output transition, and the input transition that we are trying to justify for this output transition. Columns 3-6 show the values that the on-path input (A) and off-path input (B) must take in the first and second cycles to justify the desired transition. The final column shows the formula to compute $diffj$ in terms of the controllability of the inputs. 22

3.2 Computation of $diffp$ for different gate types. In the case of 2-input gates, we assume without loss of generality that input A is the on-path input and B is the off-path input. The first two columns show the output transition, and the input transition that we are trying to propagate for this on-path input transition. Columns 3-6 show the values that the output (X) and off-path input (B) must take in the first and second cycles to propagate the desired transition. The final column shows the formula to compute $diffp$ in terms of the controllability of the off-path input and observability of output. 23

3.3 Probability of exceeding the nominal critical path delay in a 32×32 Wallace Tree Multiplier after adding delay to the rare path. When uniformly distributing the delay over the path, the longest delay exceeds 2520 ps for 57 of 200,000 random applied vectors. After using genetic algorithm (Sec. 3.4) to distribute the delay, the circuit delay never exceeds 2520 ps in 260 million random vectors. 29

3.4 Attack complexity of the proposed improved Bug Attack using the Trojan Multiplier assuming a 256 bit curve. 35

4.1 Performance figure of our PRESENT-80 encryption designs. 58

4.2 Area comparison (post-layout) of PRESENT TI implementation with and without inserted Trojan (realized by delay gates). 68

4.3 Frequency ranges for the different design states. 68

5.1 Most likely 15-bit patterns 90

5.2 NIST test suite results for Trojan free and Trojan infected TRNG 91

LIST OF FIGURES

Figure **Page**

3.1 Flowchart of the proposed method for creating a stealthy PDF (path delay faults). 15

3.2 Propagating an input transition to an output transition requires current to charge or discharge a capacitor. Decreasing width or increasing length of MOSFETs are two ways of reducing switching current and increasing propagation delay. 18

3.3 Circuit and corresponding timing graph. 20

3.4 Fault simulation of rare path and 750 random paths of 32×32 Wallace tree multiplier. 27

3.5 Error probability of circuit before and after optimizing delay assignment of rare path and 9 other paths in a 32×32 Wallace tree multiplier. 27

3.6 Increasing the rare path delay increases the probability of causing an error when random vectors are applied. This delay is allocated to gates according to the delay distribution algorithm. The results are shown for different clock periods. 30

4.1 Exemplary TI construction with a correction term C. 42

4.2 Status of the design with Trojan at different clock frequencies. 43

4.3 Two routes of the same signal in a Spartan-6 FPGA, manually perfromed by `FPGA Editor`. 52

4.4 Design architecture of the PRESENT TI as the case study. 56

4.5 Block diagram of the noise generator. 59

4.6 PRNG off, clock 168 MHz (Trojan not triggered), (top) a sample power trace, t-test results (right) with 100,000 traces, (left) absolute maximum over the number of traces. 61

4.7 PRNG on, clock 168 MHz (Trojan not triggered), t-test results (right) with 100,000,000 traces, (left) absolute maximum over the number of traces. 62

4.8 PRNG on, clock 216 MHz (Trojan triggered), (top) a sample power trace, t-test results (right) with 100,000,000 traces, (left) absolute maximum over the number of traces. 64

4.9 PRNG on, clock 216 MHz (Trojan triggered), 50,000,000 traces, DPA attack result targeting a key nibble based on an S-Box output bit at the first round. 65

4.10 Layout schematic and photos of the ASIC prototypes. 65

4.11 90 nm ASIC, PRNG off, clock frequency 25 MHz (trojan not triggered), t-test results with 1 million traces (left), absolute maximum t-value over the number of traces (right). 70

4.12 90 nm ASIC, PRNG on, clock frequency 25 MHz (trojan not triggered), t-test results with 50 million traces (left), absolute maximum t-value over the number of traces (right). 71

4.13 90 nm ASIC, PRNG on, clock frequency 85 MHz (trojan triggered), t-test results with 50 million traces (left), absolute maximum t-value over the number of traces (right). 72

4.14 90 nm ASIC, PRNG on, clock frequency 85 MHz (trojan triggered), CPA results targeting a key nibble based on an S-Box output bit with 25 million traces (right), absolute maximum correlation coefficient over the number of traces (left). 73

4.15 65 nm ASIC, PRNG off, clock frequency 25 MHz (Trojan not triggered), t-test results with 1 million traces (left), absolute maximum t-value over the number of traces (right). 74

4.16 65 nm ASIC, PRNG on, clock frequency 25 MHz (Trojan not triggered), t-test results with 80 million traces (left), absolute maximum t-value over the number of traces (right). 74

4.17 65 nm ASIC, PRNG on, clock frequency 50 MHz (Trojan triggered), t-test results with 80 million traces (left), absolute maximum t-value over the number of traces (right). 75

4.18 65 nm ASIC, PRNG on, clock frequency 50 MHz (trojan triggered), CPA results targeting a key nibble based on an S-Box output bit with 40 million traces (right), absolute maximum correlation coefficient over the number of traces (left). 76

5.1 TRNG system block diagram [86] 79

5.2 3-edge ring oscillator 79

5.3 Output waveforms of the regular RO (bottom) and 3-edge RO (top) 79

5.4 Threshold voltage manipulation of the 3-edge RO 83

5.5 Output waveform of the Trojan infected 3-edge RO at (left) 25°C where there is a large time to collapse, and (right) 120°C where there is a very small time to collapse 84

5.6 Jitter effect on the Trojan infected TRNG counter values 86

5.7 Distribution of 3-edge RO cycles to collapse at different environmental temperatures 92

5.8 Output patterns of (a) the Trojan free TRNG, and (b) the Trojan infected TRNG, raster scanning left-to-right then top-to-bottom. 92

5.9 Output values of (a) Trojan free TRNG and (b) Trojan infected TRNG at 120°C. 93

CHAPTER 1

MOTIVATION

1.1 Introduction

Cryptographic primitives are often the most trusted components in modern security solutions, ranging from network routers to IoT devices. Unfortunately, this makes cryptographic algorithms an attractive target for subversion by malicious actors. Manipulating hardware implementations as opposed to software implementations can lead to cryptographic Trojans that are particularly difficult to detect. It is widely believed that such Trojans are of special interest to nation-state adversaries. Hardware Trojans are malicious alterations of the physical design that compromise the security or safety of the attacked device. They have gained increasing attention in academia, industry and government agencies over the last ten years or so. There is a large body of research concerned with various methods for detecting Trojans, cf., e.g., [42, 87, 24, 9]. On the other hand, there is scant treatment in literature about how to design Trojans. Nevertheless, Trojan detection and design are closely related: in order to design effective detection mechanisms and countermeasures, we need an understanding of how Hardware Trojans can be built. This holds in particular with respect to Trojans that are specifically designed to avoid detection. The situation is akin to the interplay of cryptography and cryptanalysis.

There are several different ways that hardware Trojans can be inserted into an IC [42]. The insertion scenarios that have drawn the most attention in the past are hardware Trojans introduced during manufacturing by an untrusted semiconductor foundry. One of the main motivations behind this is the fact that the vast majority of ICs world wide are fabricated abroad, and a foundry can possibly be pressured by

a government agency to maliciously manipulate the design. However, we note that a similar situation can exist in which the original IC designer is pressured by her own government to manipulate all or some of the ICs, e.g., those that are used in overseas products. Similarly, 3rd party IP cores are another possible insertion point. The possibility of hardware Trojan insertion is not restricted to the manufacturing in foreign countries. Government mandated backdoors or malicious employees could also be the source of Hardware Trojans. All of these insertion scenarios have in common that the party inserting the Trojan will have a main goal of designing/implementing the Trojan in such a way that the chance of detection becomes very low.

The primary setting we consider in this dissertation is modification during manufacturing, but the methods also carry over to the other scenarios mentioned above. The Trojan will be inserted by modifying a few gates at the sub-transistor level during manufacturing. This contribution is concerned with cryptographic Trojans which possess zero overhead in terms of logic resources and are thus, extremely stealthy. The dissertation introduces three different techniques related to hardware Trojans including i) a design methodology for inserting a stealthy parametric hardware Trojan, ii) a side-channel hardware Trojan for provably-secure SCA protected implementations, iii) a temperature-based hardware Trojan for ring oscillator-based TRNGs.

In our design methodology for inserting hardware Trojans, the goal is to select an extremely rare path and chose the delays of its gates such that only for extremely rare input combinations these delays add up to a path delay fault. Since not a single transistor is removed or added to the design and the changes to the individual gates are minor, the Trojan is very difficult to detect post-manufacturing using reverse-engineering, visual inspection, side-channel profiling or most other known detection methods. Due to the extremely rare trigger conditions, it is also highly unlikely that the Trojan will be detected during functional testing. Even full reverse-engineering of the IC will not reveal the presence of the backdoor. Similarly, since the actual Trojan will be inserted in the last step of the design flow, the Trojan will not be present at

higher abstraction levels such as the netlist. Accordingly, this type of Trojan is also very interesting for the scenario of stealthy, government-mandated backdoors. The number of engineers that are aware of the Trojan would be reduced to a minimum since even the designers of the Trojan-infected IP core would not be aware that such a backdoor has been inserted into the product. This can be crucial to eliminate the risk of whistle blowers revealing the backdoor. In summary, our methodology overcomes two major problems a Trojan designer faces, namely making the Trojan detection resistant and to provide a very rare trigger condition.

Besides the hardware Trojan design methodology, this dissertation focuses on an aspect of hardware security that is one of the major research areas in hardware security: side-channel analysis. In a side-channel analysis an attacker exploits the fact that an embedded device is not a black-box that obtains defined inputs and only produces defined outputs. Instead, every physical system that performs some kind of computation will inevitably leak additional information over physical channels, such as the power consumption, the required execution time, or the thermal profile. In side-channel analysis, these physical properties are first measured and then exploited to derive additional information about the embedded system. Such information could be the secret key that is used in an encryption algorithm. We present a mechanism which shows how stealthy side-channel hardware Trojans can be inserted in provably-secure side-channel analysis protected implementations. Once the Trojan is triggered, the malicious design exhibits exploitable side-channel leakage leading to successful key recovery attacks. Integrating an SCA Trojan into an SCA-protected design is challenging if the device is supposed to be evaluated by a third-party certification body. To pass certification, the the device should provide the desired SCA protection under a white-box scenario, i.e., all design details including the netlist are known to the evaluation lab. We present a mechanism to design a provably- and practically-secure SCA-protected implementation which can be turned into an unprotected implementation by a Trojan adversary. In ASIC platforms, it is indeed inserted by

subtle manipulations at the sub-transistor level to modify the parameters of a few transistors. The same is achievable on FPGA applications by changing the routing of particular signals, without other resource utilization overhead. The underlying concept is based on a secure masked hardware implementation which does not exhibit any detectable leakage. However, by running the device at a particular clock frequency one of the requirements of the underlying masking scheme is not fulfilled anymore, i.e., the Trojan is triggered, and the device's side-channel leakage can be exploited.

High entropy random numbers are an essential component in many facets of information security, which forms the foundation for many cryptographic algorithms used to build cryptosystems. Some common applications are generating private keys, nonces, random numbers in challenge response protocols, and random numbers in countermeasure implementations to mask key-dependent values. One of the most popular methods for generating random numbers is sampling jittery signals generated by ring oscillators (ROs) [86] and [25]. In this dissertation, we present a parametric hardware Trojan for an RO-based TRNG presented in [86] in such a way that it works correctly under normal environmental conditions, but it produces non-random and predictable outputs at particular environmental conditions such as high environmental temperature. Our Trojan does not require the addition of any additional logic (even a single gate) to the design, making it extremely hard to detect. More precisely, our technique injects a *parametric* Trojan that can be triggered. Under normal conditions the randomness of the TRNG output is not affected, which enables the Trojan to avoid being detected by an evaluation lab. By increasing the temperature of the subverted device (or by increasing its workload) the Trojan is triggered and exhibits non-random and periodic outputs. We show that by injecting this Trojan, we are able to control the output of the TRNG. This biasing significantly lowers the security level even of highly protected crypto-core implementations rely on the TRNG. Also we elaborate a stochastic model based on Markov Chains by which the attacker's knowledge enables predicting the output of the Trojan infected TRNG.

1.2 Organization and Contribution of the Thesis

The main contributions in this thesis are as follows:

- The thesis introduces a new class of parametric hardware Trojans, the Path Delay Trojans. They possess the two desirable features that they are (i) very stealthy and thus difficult to detect with most standard methods and (ii) have very rare trigger conditions. We present an automation flow for inserting the proposed style of Trojan. We propose an efficient, SAT solver-based path selection algorithm, which identifies suitably rare paths in a given target circuit. We also propose a second algorithm, based on genetic algorithms, for distributing the necessary delay along the rare path to minimize its impact on the remaining circuit. As a case study for the effectiveness of the proposed method, a Trojan multiplier is designed. We were able to identify a rare path and perform specific delay modification in a 32-bit multiplier circuit model in such a way that the faulty behavior only occurs for very few combinations of two consecutive input values. We note that the input space of the multiplier is $(2^{32})^2 = 2^{64}$ so that random input values occur very rarely during regular operation. We show how the Trojan multiplier can used for realizing the bug attack by Biham et al. [10, 11] and propose a related attack on the ECDH key agreement protocol. We show that the attacker can engineer the failure probability to the desired level by increasing the introduced propagation delay of the Trojan.

- The thesis presents a mechanism to design a provably- and practically-secure SCA-protected implementation which can be turned into an unprotected implementation by a Trojan adversary. Our Trojan does not add any logic (even a single gate) to the design, making it very hard to detect. In case of ASIC platforms, the trojan is added by slightly changing the characteristic of a few transistors, and for FPGA platforms by changing the routing of particular signals. Most notably, our technique is **not** based on the leakage of the PRNG,

and it does not affect the provable-security feature of the underlying design unless the Trojan is triggered. Under normal conditions the device does not exhibit any SCA leakage. By increasing the clock frequency of the malicious device (or by decreasing its supply voltage) the Trojan is triggered and exhibits exploitable leakage. The high clock frequency that triggers the Trojan is beyond the maximum frequency at which the device can correctly operate. Hence, the device is not expected to be evaluated under such a condition by evaluation labs.

- The thesis proposes parametric hardware Trojans for RO-based TRNGs. We target the TRNG presented in [86] so that it works correctly under normal environmental conditions, but produces predictable outputs at a particular high temperature. Our parametric Trojan does not require the addition of any logic to the design, making it extremely hard to detect. We show that by injecting this Trojan, we are able to controllably bias the output of the TRNG, and we elaborate a stochastic Markov Chain model by which the attacker's knowledge of the Trojan enables predicting the outputs of the Trojan infected TRNG. This biasing can compromise the security of any functionality that relies on the TRNG.

The chapters of the thesis are structured as follows. Chapter 2 deals with necessary background and definitions in the areas of hardware Trojans and threshold implementations as an SCA countermeasure, and reviews related work in these areas. Chapter 3 introduces our methodology to design stealthy parametric hardware Trojans that induce path delay faults (PDF) for extremely rare inputs. The chapter presents path selection and delay distribution algorithms, applies them to a 32-bit multiplier circuit, and shows how to exploit the specific fault model of the path delay Trojan multiplier to attack ECDH key agreement protocols. Chapter 4 shows how to insert a parametric hardware Trojan with very low overhead into SCA-resistant designs on ASIC and FPGA platforms to leak exploitable information through side channels. Chapter 5

explain how we insert a stealthy parametric hardware Trojan into the RO-based TRNG circuit which will be triggered by a specific high operating temperature, and we elaborate a stochastic model based on Markov Chains by which the attacker can predicted the output of the Trojan infected TRNG. The findings in this thesis are summarized in Chapter 6 and possible future work is discussed.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Hardware Trojans

The power of hardware Trojans was first demonstrated by King et al. in 2008 by showing how a hardware Trojan inserted into a CPU can enable virtually unlimited access to the CPU by an external attacker [46]. The Trojan presented by King et el. was inserted into the HDL code of the design. Similarly, Lin et el. presented a hardware Trojan that stealthily leaks out the cryptographic key using a power side-channel [51]. This hardware Trojan was also inserted at the HDL or netlist level, similarly to the hardware Trojans that were designed as part of a student hardware Trojan challenge at ICCD 2011 [71]. How to build stealthy Trojans at the layout-level was demonstrated in 2013 by Becker et el. which showed how a hardware Trojan can be inserted into a cryptographically secure PRNG or a side-channel resistant S-Box only by manipulating the dopant polarity of a few registers [7]. Another idea proposed in literature is building hardware Trojans that are triggered by aging [79]. Such Trojans are inactive after manufacturing and only become active after the IC has been in operation for a long time. Kumar et el. proposed a parametric Trojan [50] that triggers probabilistically with a probability that increases under reduced supply voltage.

Compared to research concerned with the design of hardware Trojan, considerably more results exist related to different hardware Trojan detection mechanisms and countermeasures. Most research focuses on detecting hardware Trojans inserted during manufacturing. In many cases, a Trojan-free golden model serves as a reference. One important question is how to get to a Trojan-free golden model. One approach

proposed is to use visual reverse-engineering of a few chips to ensure that these chips were not manipulated. For this the layout is compared to SEM images of the chip. In [4] methods of how to automatically do this are discussed. Please note that that not all hardware Trojans are directly visible in black-and-white SEM images. For example, to detect the dopant-level hardware Trojans additional steps are needed, e.g., the method presented by Sugawara et el. [80]. One motivation of our work is that we might achieve an even higher degree of stealthiness by only slowing down transistors as opposed to completely changing transistors as has been done in [7]. Such parametric changes can be done cleverly to make visual reverse-engineering very difficult as discussed in Section 3.2. Another approach to Trojan detection uses power profiles that are used to compare the chip-under-test with previously recorded side-channel measurement of the golden chip. The most popular approach uses power side channels, first proposed by Agrawal et el. [3]. The idea to build specific Trojan detection circuitry has also been proposed, e.g., in [72]. However, these approaches usually suffer from the problem that a Trojan can also be inserted into such detection circuitry. Preventing hardware Trojans inserted at the HDL level by third party IP cores has been discussed, e.g., in [41] and [84]. Efficient generation of test patterns for hardware Trojans triggered by rare input signals is the focus of work by Chakraborty et el. [24] and Saha et el. [74]. [45] and [44] focus on preventing a reverse engineer from learning the exact function implemented on a target chip.

Closely related to hardware Trojans are certain types of physical attacks. A physical attack on random number generators was presented in [58] which targets an RO based TRNG implemented in an IC. Injecting a sine wave onto the power supply, the operating conditions were modified and a bias appeared at the output signal. Another physical attack presented in [6], targets another RO based TRNG [85] using an electromagnetic attack. In this attack, the ROs were locked on the injection frequency, generating a controllable bias at the output. The work in [59] investigated

the impact of power and clock glitches, temperature and underpowering on a TRNG design [25] implemented on an FPGA.

2.2 Side-Channel Based Trojans

Our focus in this subsection is Trojans which leak out secrets through a side channel. The first such Trojan has been introduced in [52] and [53] which stealthily leaks out the cryptographic key through a power consumption side channel. This Trojan, made by a moderately large circuit including an LFSR and leaking circuit, is inserted at the HDL or netlist level. Therefore, it is likely detected by a Trojan inspector. Further, the Trojan designs in these works [52, 53] are not triggerable, i.e., they always leak through the side channel, which might be exploited by anybody not only the Trojan attacker.

On the other hand, the cryptographic devices— if pervasive and/or ubiquitous— are in danger of side-channel analysis (SCA) attacks. Two decades after the introduction of such physical attacks [47, 48], integration of dedicated SCA countermeasures is a must for devices which deal with security. Therefore, if the design is not protected against SCA threats, any SCA adversary would be able to reveal the secrets independent of the existence of such a Trojan [53].

In a follow-up work [43], the authors expressed a relatively similar concept on an SCA-protected implementation. Their technique is based on inserting a logical circuit forming an LFSR-based Trojan leaking the internal state of a PRNG. As a side note, random number generators are necessary modules for those SCA-protected implementations which are based on masking [56]. Hence, the Trojan adversary would detect the internal state of the PRNG by means of SCA leakages and can then be able to conduct DPA attacks due to knowing the masks. It should be noted that those products which need to be protected against physical attacks are usually evaluated by a third-party certification body, e.g., through a common criteria evaluation lab.

Therefore, due to its relatively large circuit, such a Trojan is very likely detected by an inspector.

As another work in this domain, we should refer to [7], where the Trojan is inserted by changing the dopant polarity of a few transistors in a circuit realized by the DPA-resistant logic style iMDPL [67]. However, no such logic styles can perfectly provide security, and the leakage of an iMDPL circuit can still be exploited by ordinary SCA adversaries [60].

2.3 Threshold Implementation

It can be said that *masking* is one of the most-studied countermeasures against SCA attacks. Masking is based on the concept of *secret sharing*, where a secret $\boldsymbol{x}$ (e.g., intermediate values of a cipher execution) is represented by a number of shares $(\boldsymbol{x}^1, \ldots, \boldsymbol{x}^n)$. In case of an (n, n)-threshold secret sharing scheme, having access to $t < n$ does not reveal any information about $\boldsymbol{x}$. Amongst those is Boolean secret sharing, known as Boolean masking in the context of SCA, where $\boldsymbol{x} = \bigoplus_{i=1}^{n} \boldsymbol{x}^i$. Hence, if the entire computation of a cipher is conducted on such a shared representation, its SCA leakage will be (on average) independent of the secrets as long as no function (e.g., combinatorial circuit) operates on all n shares.

Due to the underlying Boolean construction, application of a linear function $\mathcal{L}(.)$ over the shares is straightforward since $\mathcal{L}(\boldsymbol{x}) = \bigoplus_{i=1}^{n} \mathcal{L}(\boldsymbol{x}^i)$. All the difficulties belong to implementing non-linear functions over such a shared representation. This concept has been applied in hardware implementation of AES (mainly with $n = 2$) with no success [66, 57, 22, 61] until the Threshold Implementation (TI) – based on sound mathematical foundations – has been introduced in [65], which defines minimum number of shares $n \geq t + 1$ with t the algebraic degree of the underlying non-linear function. For simplicity (and as our case study is based on) we focus on quadratic Boolean functions, i.e., $t = 2$, and minimum number of shares $n = 3$. For example for cubic bijection functions, decomposition techniques can be used to obtain quadratic

bijection functions [17, 30, 31]. Suppose that the TI of the non-linear function $y = \mathcal{F}(\boldsymbol{x})$ is desired, i.e., $(\boldsymbol{y}^1, \boldsymbol{y}^2, \boldsymbol{y}^3) = \mathcal{F}^*(\boldsymbol{x}^1, \boldsymbol{x}^2, \boldsymbol{x}^3)$, where

$$\boldsymbol{y}^1 \oplus \boldsymbol{y}^2 \oplus \boldsymbol{y}^3 = \mathcal{F}(\boldsymbol{x}^1 \oplus \boldsymbol{x}^2 \oplus \boldsymbol{x}^3). \tag{2.1}$$

Indeed, each output share $\boldsymbol{y}^{i\in\{1,2,3\}}$ is provided by a component function $\mathcal{F}^i(.,.)$ which receives only two input shares. In other words, one input share is definitely missing in every component function. This, which is a requirement defined by TI as *non-completeness*, supports the aforementioned concept that "no function (e.g., combinatorial circuit) operates on all n shares", and implies the given formula $n \geq t+1$. Therefore, three component functions $(\mathcal{F}^1(\boldsymbol{x}^2, \boldsymbol{x}^3), \mathcal{F}^2(\boldsymbol{x}^3, \boldsymbol{x}^1), \mathcal{F}^3(\boldsymbol{x}^1, \boldsymbol{x}^2))$ form the shared output $(\boldsymbol{y}^1, \boldsymbol{y}^2, \boldsymbol{y}^3)$.

2.3.1 Uniformity

In order to fulfill the above-given statement that "having access to $t < n$ shares does not reveal any information about $\boldsymbol{x}$", the shares need to follow a uniform distribution. For simplicity suppose that $n = 2$, and the shares $(\boldsymbol{x}^1, \boldsymbol{x}^2)$ represent secret $\boldsymbol{x}$. If the distribution of $\boldsymbol{x}^1$ has a bias (i.e., not uniform) which is known to the adversary, he can observe the distribution of $\boldsymbol{x}^2 = \boldsymbol{x} \oplus \boldsymbol{x}^1$ and guess $\boldsymbol{x}$. Hence, the security of masking schemes[1] relies on the uniformity of the masks. More precisely, when $\boldsymbol{x}^1 = \boldsymbol{m}$, $\boldsymbol{x}^2 = \boldsymbol{x} \oplus \boldsymbol{m}$, and $\boldsymbol{m}$ is taken from a randomness source (e.g., a PRNG), the distribution of $\boldsymbol{m}$ should be uniform (with full entropy).

The same holds for higher-order masking, i.e., $n > 2$. However, not only the distribution of every share but also the joint distribution of every $t < n$ shares is important. In case of $\mathcal{F}^*(.,.,.)$ as a TI of a bijective function $\mathcal{F}(.)$, the *uniformity* property of TI is fulfilled if $\mathcal{F}^*(.,.,.)$ forms a bijection. Otherwise, the security of such an implementation cannot be guaranteed. Note that fulfilling the uniformity

[1]Except those which are based on low-entropy masking [23, 55].

property of TI constructions is amongst its most difficult challenges, and it has been the core topic of several articles [17, 68, 14, 65, 8]. Alternatively, the shares can be remasked at the end of every non-uniform shared non-linear function (see [14, 62]), which requires a source to provide fresh randomness at every clock cycle. Along the same line, another type of masking in hardware (which reduces the number of shares) has been developed in [73, 36], which (almost always) needs fresh randomness to fulfill the uniformity.

We should emphasize that the above given expressions illustrate only the first-order TI of bijective quadratic functions. For other cases including higher-order TI we refer the interested reader to the original articles [65, 14, 17].

CHAPTER 3

A DESIGN METHODOLOGY FOR STEALTHY PARAMETRIC HARDWARE TROJANS

[1]In this work, we examine how particularly stealthy Trojans can be introduced to a given target circuit. The Trojans are triggered by violating the delays of very rare combinational logic paths. These are parametric Trojans, i.e., they do not require any additional logic and are purely based on subtle manipulations on the sub-transistor level to modify the parameters of the transistors. The Trojan insertion is based on a two-phase approach. In the first phase, a SAT-based algorithm identifies rarely sensitized paths in a combinational circuit. In the second phase, a genetic algorithm smartly distributes delays for each gate to minimize the number of faults caused by random vectors.

As a case study, we apply our method to a 32-bit multiplier circuit resulting in a stealthy Trojan multiplier. This Trojan multiplier only computes faulty outputs if specific combinations of input pairs are applied to the circuit. The multiplier can be used to realize bug attacks, introduced by Biham et al. [10, 11]. In addition to the bug attacks proposed previously, we extend this concept for the specific fault model of the path delay Trojan multiplier and show how it can be used to attack ECDH key agreement protocols.

---

[1]The research presented in this chapter was published in [32].

3.1 Overview

This work implements Trojan functionality in a given target circuit by using path delay faults (PDF), without modification to logic circuit, to induce inaccurate results for extremely rare inputs. Before describing the details of our method, we first define the notion of a viable delay-based Trojan in the unmodified HDL of the circuit as follows. A viable delay-based trojan must posses the following two properties.

Triggerability For secret inputs, which are known to the attacker, cause an error with certainty or relatively high probability.

Stealthiness For randomly chosen inputs, cause an error with extremely low probability.

As shown in Fig. 3.1, our method of creating triggerable and stealthy delay-based Trojans consists of two phases: *path selection* and *delay distribution*. We give an overview of each phase here, and give detailed descriptions in sections 3.3 and 3.4.

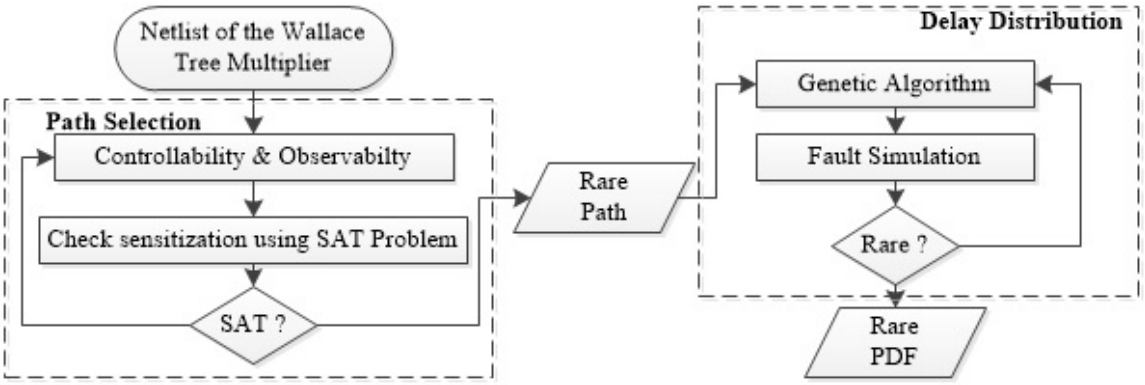


Figure 3.1: Flowchart of the proposed method for creating a stealthy PDF (path delay faults).

Path Selection: The path selection phase finds a rarely sensitized path from the primary inputs of a combinational circuit to the primary outputs. The algorithm chooses the path incrementally by adding gates to extend a subpath backward toward inputs and forward toward outputs. The selection of which gates to include is guided

by controllability and observability metrics so that the path will be rarely sensitized. To ensure that the selected path can be triggered, a SAT-based check is performed to ensure that the path remains sensitizable each time a gate is added. In addition to ensuring that the path is sensitizable, the SAT-based check also provides the Trojan designer with a specific input combination that will sensitize the path. This input combination will later serve as the trigger for the Trojan. Details of the path selection are given in Sec. 3.3.

Delay Distribution: After a rarely sensitized path is selected, the overall delay of the path must be increased so that a delay fault will occur when the path is sensitized; this is required for the Trojan to be triggerable. However, any delay added to gates on the selected path may also cause delay faults on intersecting paths, which would cause more frequent errors and compromise stealthiness. Our delay distribution heuristic addresses this problem by smartly choosing delays for each gate to minimize the number of faults caused by random vectors. At the same time, the approach ensures that the overall path delay is sufficient for the fault to occur when the trigger vectors are applied. Details of delay distribution are given in Sec 3.4.

3.2 Delay Insertion

Delay faults occur when the total propagation delay along a sensitized circuit path exceeds the clock period. Our algorithm causes delay faults by increasing the delay of gates on a chosen path. While the approach is compatible with any mechanism for controlling gate delays, in this section we provide background on practical methods that a Trojan designer might use to implement slow gates. In static CMOS logic, a path delay fault is not triggered by a single input vector, but instead is triggered by a sequence of two input vectors applied on consecutive cycles. The physical reason for delay being caused by a pair of inputs is that delay depends on the charging or discharging of capacitances, and the initial states of these capacitances in the second vector are determined as final states from the first vector. Assuming the capacitances

need to be charged or discharged along a path, as is the case in delay faults, the delay of each gate depends on how quickly it can charge or discharge some amount of capacitance on its output node, and diminishing the ability of a gate to do so will slow it down. There are several stealthy ways of changing a circuit to make gates slower. As an example, we list three methods below. We note that circuit designers typically face the opposite and considerably more difficult task, namely making gates fast. The ever-shrinking feature size of modern ICs is amenable to our goal of slowing gates down through minuscule alterations.

3.2.1 Decrease Width

A gate library typically includes several drive strengths for each gate type, corresponding to different transistor widths. A narrow transistor is slower to charge a load capacitance because transistor current is linear in channel width. A straightforward way to increase delay is to replace a gate with a weaker variant of the same gate, or to create a custom cell variant with an extremely narrow channel. A limitation to using a downsized gate is that an attacker who delayers the chip could potentially observe the sizing optically, depending on how much the geometry has been altered.

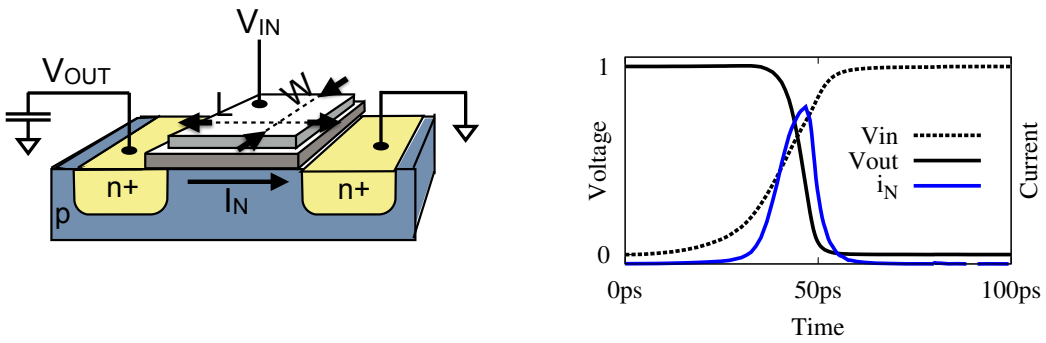
3.2.2 Raise Threshold

A second way of increasing gate delay is to increase threshold voltages of selected transistors through doping or body biasing. Dual-Vt design is common in ICs and allows transistors to be designated as high or low threshold devices; low threshold devices are fast and used where delay is critical, and high threshold devices are slow and used elsewhere to reduce static power. Typically no more than two threshold levels are used on a single chip because creating multiple thresholds through doping requires additional process steps, but in principle an arbitrary number of thresholds can be created. Body biasing, changing the body-source voltage of MOSFETs, is another way to change threshold and delay [49]; specifically, a reverse body bias (i.e., body terminal at lower voltage than source) raises threshold voltage and slows down

a device. Regardless whether the mechanism is doping or body biasing, a raised threshold voltage will cause transistors to turn on later when an input switches, and to conduct less current when turned on, so the output capacitance connected to the transistor will be charged or discharged more slowly. Both, changing to dopant concentrations and body biasing, are difficult to detect, even with invasive methods.

3.2.3 Increase Gate Length

Delay of chosen gates can be increased by gate length biasing. Lengthening the gate of transistor causes a reduction in current, and therefore increases delay [39]. Again, the likelihood of detection depends on the degree of the alteration.



(a) Annotated NMOS transistor

(b) Switching Event

Figure 3.2: Propagating an input transition to an output transition requires current to charge or discharge a capacitor. Decreasing width or increasing length of MOSFETs are two ways of reducing switching current and increasing propagation delay.

We note that the methods sketched above (and other slow-down alterations) can be combined such that each manipulation is relatively minor and, thus, more difficult to detect.

3.3 Phase I: Rare Path Selection

Fundamentally, the challenge in designing and validating triggerable and stealthy delay-based Trojans is that timing and logical sensitization cannot be decoupled.

Regardless of the type of path sensitization considered, the probability of causing an error is not a well-defined concept until after delays are assigned. Therefore, when designing a candidate Trojan, path selection and delay assignment must both be considered. We will use a heuristic for this which combines logical path selection and delay distribution along a chosen path.

In this phase we try to select a path among huge number of paths existing in the netlist of a multiplier circuit, in such a way that random inputs will very rarely sensitize the path. The rareness is a first step towards ensuring stealthiness of the Trojan.

3.3.1 Controllability and Observability

Before giving our algorithm, we introduce several preliminaries. First, we note that every node in the circuit has a controllability metric and an observability metric associated with the 0 value and the 1 value. Controllability and observability are common metrics used in testing. Controllability of a 0 or 1 value on a circuit node is an estimate of the probability that a random input vector would induce that value on that node. Observability of a 0 or 1 value on a node is an estimate of the probability with which that value would propagate to some output signal when a random vector is applied. For rareness, we seek a path that includes low controllability nodes and low observability nodes, as this would indicate that the values on the path rarely occur randomly, and when they do occur they are usually masked from reaching the outputs. We estimate controllability using random simulation, and observability using random fault injection [40].

3.3.2 Timing Graph

The propagation delays of logic paths in combinational VLSI circuits are typically represented using weighted DAGs called timing graphs. Each node in a circuit will have two nodes in the timing graph, representing rising and falling transitions on the node; we use the terms transition and node interchangeably when discussing timing

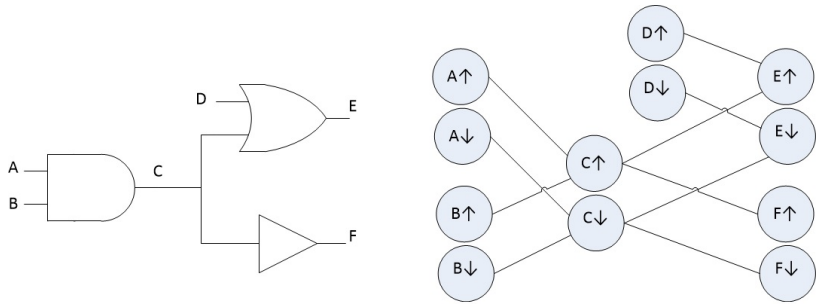


Figure 3.3: Circuit and corresponding timing graph.

graphs. A directed edge between two nodes exists if the transition at the tail of the edge can logically propagate to the one at the head. The edges that exist in the timing graph therefore depend on the logic function of each gate in the circuit (see Fig. 3.3).

For example, an AND gate with inputs A and B, and output X, will have an edge from $A \uparrow$ to $X \uparrow$, from $A \downarrow$ to $X \downarrow$, from $B \uparrow$ to $X \uparrow$, and from $B \downarrow$ to $X \downarrow$, but will not have an edge from $A \uparrow$ to $X \downarrow$ because a rising transition on an AND gate input cannot induce a falling output. In timing analysis, e.g. STA, the edge weights of a timing graph represent propagation delay, but for our purpose of path selection, the delays are ignored and we utilize only the connectivity of the timing graph.

3.3.3 Selecting a Path Through Timing Graph

Our path selection technique seeks to find a path π through the timing graph of the circuit that is rarely sensitized. Note that the delays are not considered in this phase of the work. Path π is initialized to contain a single hard to sensitize transition somewhere in the middle of the circuit. More formally, the starting point for the path search is a rising or falling transition on a single node such that the product of its 0 and 1 controllability values is the lowest among all nodes in the circuit. This initial single-node path π is then extended incrementally backward until reaching the primary inputs (PIs), and extended incrementally forward until reaching the primary outputs (POs). The backward propagation is given in Alg. 1, and the forward propagation is given in Alg. 2.

First we explain the backward propagation heuristic in Alg. 1. Starting from the first transition (i.e. the tail) on the current path π, we repeatedly try to extend the path back toward the PIs by prepending one new transition to the path. To select such a transition, the algorithm creates a list of candidate transitions that can be be prepended to the path. In the timing graph, these candidates are predecessor nodes to the current tail of π. The list of candidate nodes is then sorted according to $diffj$, the difficulty of creating the necessary conditions to justify the transition. Tab. 3.1 shows the formula used to compute $diffj$ for each transition on each gate type. Note that our difficulty metric is weighted to always prefer robust sensitization first, and only resort to non-robust sensitization when there are no robustly sensitizable nodes in the list of candidates. Whenever a node is prepended to π to create a candidate path π' (line 5) the sensitizability of π' is checked by calling *check-sensitizability* function. In this function SAT-based techniques [26] are used to check sensitizability of a path and to find a vector pair that justifies and propagates a transition along the path (line 6). If the SAT solver returns SAT, then path π' is known to be a subpath of a sensitizable path from PIs to POs. Because the candidates are visited in order of preference, there is no need to check other candidates after finding a first candidate that produces a sensitizable path. At this point, the algorithm updates π to be π' and the algorithm exits the for loop having extended the path by one node. If this newly added tail node is not a PI, then the algorithm will again try to extend it backwards.

The forward propagation algorithm (Alg. 2) is similar to the aforementioned backward propagation algorithm, except that it adds nodes to the head of the path until reaching POs. At each step of the algorithm, a list of candidates is again formed. In this case, the candidates are successors of the head of the path (line 2) instead of predecessors of the tail, and they are ordered according to difficulty of propagation (line 3) instead of difficulty of justification (See Tab. 3.2). Each time a new candidate path is created by adding a candidate node to the existing path, a sat check is again

Table 3.1: Computation of $diffj$ for different gate types. In the case of 2-input gates, we assume without loss of generality that input A is the on-path input and B is the off-path input. The first two columns show the output transition, and the input transition that we are trying to justify for this output transition. Columns 3-6 show the values that the on-path input (A) and off-path input (B) must take in the first and second cycles to justify the desired transition. The final column shows the formula to compute $diffj$ in terms of the controllability of the inputs.

	output trans.	input trans.	A		B		$Diffj$
			v(1)	v(2)	v(1)	v(2)	
X = AND(A,B)	$X\downarrow$	$A\downarrow$	1	0	1	1	$C_1(A) * C_0(A) * C_1^2(B)$
	$X\uparrow$	$A\uparrow$	0	1	-	1	$C_0(A) * C_1(A) * C_1(B)$
X = OR(A,B)	$X\downarrow$	$A\downarrow$	1	0	-	0	$C_1(A) * C_0(A) * C_0(B)$
	$X\uparrow$	$A\uparrow$	0	1	0	0	$C_0(A) * C_1(A) * C_0^2(B)$
X = XOR(A,B)	$X\downarrow$	$A\downarrow$	1	0	0	0	$C_1(A) * C_0(A) * C_0^2(B)$
	$X\downarrow$	$A\uparrow$	0	1	1	1	$C_0(A) * C_1(A) * C_1^2(B)$
	$X\uparrow$	$A\uparrow$	0	1	0	0	$C_0(A) * C_1(A) * C_0^2(B)$
	$X\uparrow$	$A\downarrow$	1	0	1	1	$C_1(A) * C_0(A) * C_1^2(B)$
X = BUFF(A)	$X\downarrow$	$A\downarrow$	1	0	-	-	1
	$X\uparrow$	$A\uparrow$	0	1	-	-	1
X = INV(A)	$X\downarrow$	$A\uparrow$	0	1	-	-	1
	$X\uparrow$	$A\downarrow$	1	0	-	-	1

Algorithm 1: Extend path backward to PIs while trying to maximize difficulty of justification while ensuring that path will remain sensitizable.

Require: A sensitizable subpath π in timing graph of circuit.
Ensure: A longer sensitizable subpath π in timing graph that starts at a PI

```
1: while tail(π) ∉ PIs do
2:    candidates ← (∀n|n ∈ pred(tail(π))) {transitions that can be prepended to π}
3:    candidates.order(diffj) {Order candidates by difficulty of justification}
4:    for n' ∈ candidates do
5:       π' ← (n', π) {Create a candidate path by prepending current path}
6:       if check-sensitizability(π') = SAT then
7:          π ← π' {candidate accepted, update path π with new tail}
8:          Exit for loop
9:       end if
10:   end for
11: end while
```

Table 3.2: Computation of $diffp$ for different gate types. In the case of 2-input gates, we assume without loss of generality that input A is the on-path input and B is the off-path input. The first two columns show the output transition, and the input transition that we are trying to propagate for this on-path input transition. Columns 3-6 show the values that the output (X) and off-path input (B) must take in the first and second cycles to propagate the desired transition. The final column shows the formula to compute $diffp$ in terms of the controllability of the off-path input and observability of output.

	output trans.	input trans.	X		B		$Diffp$
			v(1)	v(2)	v(1)	v(2)	
X = AND(A,B)	$X\downarrow$	$A\downarrow$	1	0	1	1	$OB_1(X) * OB_0(X) * C_1^2(B)$
	$X\uparrow$	$A\uparrow$	0	1	-	1	$OB_0(X) * OB_1(X) * C_1(B)$
X = OR(A,B)	$X\downarrow$	$A\downarrow$	1	0	-	0	$OB_1(X) * OB_0(X) * C_0(B)$
	$X\uparrow$	$A\uparrow$	0	1	0	0	$OB_0(X) * OB_1(X) * C_0^2(B)$
X = XOR(A,B)	$X\downarrow$	$A\downarrow$	1	0	0	0	$OB_1(X) * OB_0(X) * C_0^2(B)$
	$X\downarrow$	$A\uparrow$	1	0	1	1	$OB_1(X) * OB_0(X) * C_1^2(B)$
	$X\uparrow$	$A\uparrow$	0	1	0	0	$OB_0(X) * OB_1(X) * C_0^2(B)$
	$X\uparrow$	$A\downarrow$	0	1	1	1	$OB_0(X) * OB_1(X) * C_1^2(B)$
X = BUFF(A)	$X\downarrow$	$A\downarrow$	1	0	-	-	$OB_1(X) * OB_0(X)$
	$X\uparrow$	$A\uparrow$	0	1	-	-	$OB_0(X) * OB_1(X)$
X = INV(A)	$X\downarrow$	$A\uparrow$	1	0	-	-	$OB_1(X) * OB_0(X)$
	$X\uparrow$	$A\downarrow$	0	1	-	-	$OB_0(X) * OB_1(X)$

performed to ensure that the nodes are only added to π if it remains sensitizable (line 6).

3.4 Phase II: Delay Distribution

Once a path is selected, we must increase the delay of the path so that the total path delay will exceed the clock period and an error will occur when the path is sensitized. Yet, we must be careful in choosing where to add delay on the path, because the gates along the chosen path are also part of many other intersecting or overlapping paths. Any delay added to the chosen path therefore may cause errors even when the chosen path is not sensitized. To ensure stealthiness, we must minimize the probability of this by smartly deciding where to add delays along the path.

We use a genetic algorithm to decide the delay of each gate that will cause the Trojan to be stealthy. Genetic algorithm is an optimization technique that tries to

Algorithm 2: Extend path forward to POs while trying to maximize difficulty of propagation while ensuring that path will remain sensitizable.

Require: A sensitizable subpath π in timing graph of circuit.
Ensure: A longer sensitizable subpath π in timing graph that ends at a PO

```
1: while head(π) ∉ POs do
2:    candidates ← (∀n|n ∈ succ(head(π))) {transitions that can be appended to π}
3:    candidates.order(diffp) {Order candidates by difficulty of propagation}
4:    for n' ∈ candidates do
5:       π' ← (π, n') {Create a candidate path by appending to current path}
6:       if check-sensitizability(π') = SAT then
7:          π ← π' {candidate accepted, update path π with new head}
8:          Exit for loop
9:       end if
10:   end for
11: end while
```

minimize a cost function by creating a population of random solutions, and repeatedly selecting the best solutions in the population and combining and mutating them to create new solutions; the quality of each solution is evaluated according to a fitness function. We use the genetic algorithm function `ga` in Matlab [1], and do not utilize any special modifications to the genetic algorithm implementation. Our interaction with the `ga` function is limited to providing constraints that restrict the allowed solution space, and a fitness function for evaluating solutions. We describe these constraints and fitness function here.

3.4.1 Constraint on Total Path Delay

Given a chosen path π comprising gates $(p_0, p_1, \ldots, p_n)$ and assuming a target path delay of D, the genetic algorithm decides the delay of each gate on the path. Our first constraint therefore specifies that the sum of assigned delays along the path is equal to the target path delay D. To cause an error, D must exceed the clock period, and we later show advantages of using different values of D.

$$D = \sum_{i=0}^{n} d_i \tag{3.1}$$

3.4.2 Constraint on Delay of Each Gate

Next we provide the genetic algorithm with a hint that helps it to discover reasonable delays for each gate. In this step, we use d_i' to represent the nominal delay of the ith gate on chosen path π, and s_i to represent the a slack metric associated with the same gate. Each slack parameter s_i describes how much delay can be added to the corresponding gate without causing the path to exceed the clock period. Because the targeted path delay D does exceed the clock period, gate delays are allowed to exceed their computed slack. The slack for each gate is computed as a function of the nominal delay of the gate, data dependency, and the clock period [82] [29]. The following equation shows the constraint on delay of gate i, where c is a constant.

$$d_i' + s_i - c \leq d_i \leq d_i' + s_i + c \tag{3.2}$$

3.4.3 Fitness Function

Simply stated, the cost function that we want to minimize is the probability of causing an error when random input vectors are applied to the circuit. Because there is no simple closed-form expression for this, we use random simulation to evaluate the cost of any delay assignment. When the genetic algorithm in Matlab needs to evaluate the cost of a particular delay assignment, it does so by executing a timing simulator. The timing simulator, in our case ModelSim, applies random vectors to the circuit-under-evaluation and a golden copy of the circuit and compares the respective outputs to count the number of errors that occur. These errors are caused by the delay assignments in the circuit-under-evaluation. The cost that is returned from the simulator is the percentage of inputs that caused an error for this delay assignment. As the genetic algorithm proceeds through more and more generations of solutions, the quality of the solutions improve. Matlab's genetic algorithm implementation comes with a stopping criterion, so we simply allow the algorithm to run until completion.

3.5 Experimental Results

We now evaluate the effectiveness of our method of designing Trojans, using a 32×32 Wallace tree multiplier as a test case. The circuit has a nominal critical path of length 128, and the delay of this path is 2520 ps.

3.5.1 Evaluation of Phase I (Path Selection)

To evaluate the ability of our path selection algorithm (Sec. 3.3) to find a rare path, we compare the stealthiness of the path selected by the algorithm against the stealthiness of 750 randomly chosen paths. For each of these paths, we seek to find how often an error would occur under random inputs if the path delay is increased. We measure this by uniformly increasing the delay of each gate on the path such that the total delay of the path is 5040 ps, which is twice the delay of the nominal critical path. After the delay modification, 10,000 random vectors are applied and the number of error-causing vectors is counted. The histogram of Fig. 3.4 shows the result; the x-axis represents error rates, and the y-axis shows how many of the paths have each error rate. The result shows that a majority of paths would cause frequent errors if their delay is increased, and these paths are thus unsuitable for stealthy Trojans. The rare path (RP) selected by our algorithm caused an error for only 4 of 10,000 vectors. By comparison, the best of the random paths caused an error in 174 of 10,000 vectors. In this experiment, the path chosen by the path selection algorithm is 43x less likely to cause an error than the best of 750 random paths. Note that this experiment is conservative in that the amount of additional delay added is very large, and the delay is not smartly distributed along the path to minimize detection.

3.5.2 Evaluation of Phase II (Delay Distribution)

To evaluate the effectiveness of our delay distribution method, we apply the proposed method (Sec. 3.4) on 10 paths from the multiplier. These 10 paths are the rare path chosen by the path selection algorithm, and 9 paths randomly selected from the set of all paths that caused less than 10% error rates in Fig. 3.4. For each of

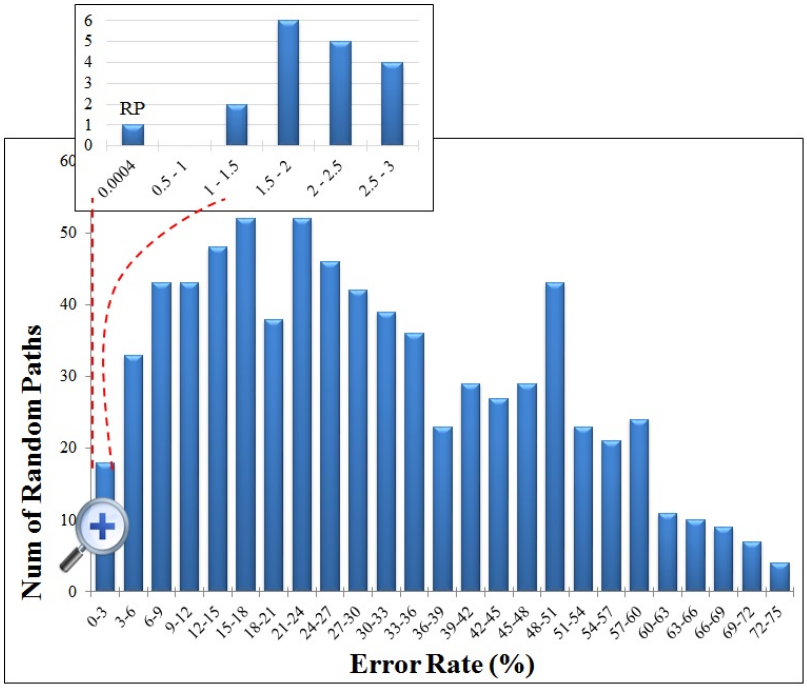


Figure 3.4: Fault simulation of rare path and 750 random paths of 32 × 32 Wallace tree multiplier.

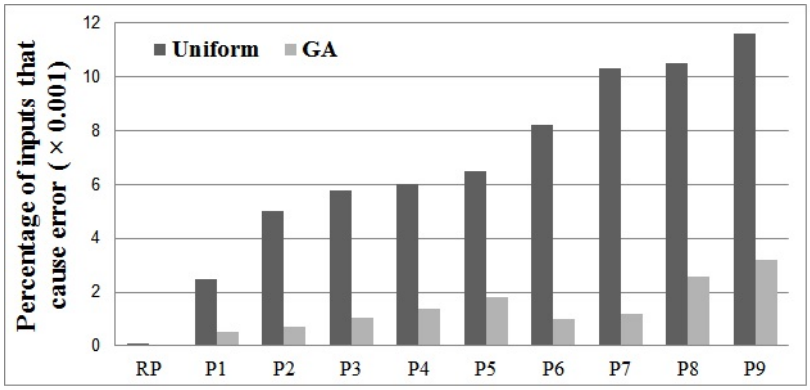


Figure 3.5: Error probability of circuit before and after optimizing delay assignment of rare path and 9 other paths in a 32 × 32 Wallace tree multiplier.

these paths, we use the genetic algorithm to optimally allocate a total delay of 3276 ps (i.e. 1.3 times of the delay of the nominal critical path) over the path, and then evaluate the error probability using random simulation with 5,000,000 vectors. Fig. 3.5 shows the error probability of each path before and after applying our proposed delay distribution method. In each case, the optimization step reduces the probability of

causing an error by at least 3.5x. For the rare path (RP), just one error in 5,000,000 vectors is caused after delay distribution. This result shows that, for a given total path delay, optimizing the delay assignment along the path can reduce the probability of having an error when random vectors are applied. It is important to note that this improvement in stealthiness comes from minimizing the side effects of the added delay, and does not impact triggerability when vectors are applied that actually sensitize the entire chosen path.

3.5.3 Overall Evaluation

We evaluate our overall methodology comprising path selection and delay distribution on the 32×32 Wallace Tree multiplier circuit. Instead of assuming a particular clock frequency, here we examine whether it is possible to add delay to the chosen rare path such that the circuit will (1) exceed the nominal critical path delay of 2520 ps when the applied input sensitizes the rare path, and (2) always have delay of less than 2520 ps otherwise. We first distribute delay uniformly over the path, and then apply the same total delay to the path but distribute it using the genetic algorithm (Sec. 3.4). The results are shown in Tab. 3.3. Despite simulating 260 million random vectors, we are unable to randomly discover any vectors in which the circuit delay exceeds 2520 ps. Yet, when applying a vector pair produced by our SAT-based sensitization check, we observe that the chosen path delay does exceed 2520 ps. As simulating 260 million vectors on a circuit this size already used more than 240 hours of computation on an AMD Opteron(TM) Processor running at 2.3GHz with 8 cores and 64 GB RAM, it will become quite expensive to check increasing numbers of vectors beyond 260 million. This highlights a significant challenge: given a space of 2^{128} possible vector pairs that might cause an error, it is very hard to estimate the probability of an error that is sufficiently rare. If the probability of error is around or above roughly 2^{-26}, then random simulation will suffice to find a few errors and estimate the error probability. If the probability of error is below roughly 2^{-98} it would be possible to use SAT to

exhaustively enumerate all 2^{30} vectors that would cause an error. Unfortunately, for very interesting region of error probabilities between 2^{-26} and 2^{-98} there is no clear solution for estimating the error probabilities.

When the amount of delay added to the rare path is increased, and the probability of error grows above 2^{-26}, the error probability can feasibly be estimated with random simulation. In this regime, we can evaluate the tradeoff of delay and trigger probability. For example, when the chosen path is given a total delay of 3150 ps allocated using genetic algorithm for delay distribution, and the circuit is operated at a clock period of 2800 ps (as might be reasonable for a nominal critical path of 2520 ps) an erroneous output occurs with probability of roughly 2^{-24} (once every 16 million multiplications) when random inputs are applied. The overall tradeoff is shown in Fig. 3.6 for different clock periods. One can exploit this tradeoff to create a desired error probability by increasing or decreasing the total amount of delay added to the chosen path.

Table 3.3: Probability of exceeding the nominal critical path delay in a 32×32 Wallace Tree Multiplier after adding delay to the rare path. When uniformly distributing the delay over the path, the longest delay exceeds 2520 ps for 57 of 200,000 random applied vectors. After using genetic algorithm (Sec. 3.4) to distribute the delay, the circuit delay never exceeds 2520 ps in 260 million random vectors.

	Delay Distribution	
	Uniform	GA
num. of times exceeding 2520 ps	57	0
num. of random vectors applied	200,000	260M
prob. of exceeding 2520 ps	0.0003	$< 2^{-26}$

3.6 Bug Attack On ECDH with a Trojan Multiplier

The main motivation of choosing a multiplier as our case study is the bug attack paper by Biham et el. [10, 11]. They showed how several public key implementations can be attacked if the used multiplier computes a faulty response for some rare inputs. The real-world implications of bug attacks were first demonstrated by Brumley et el. in 2012 when they showed how a software bug in an implementation of the reduction

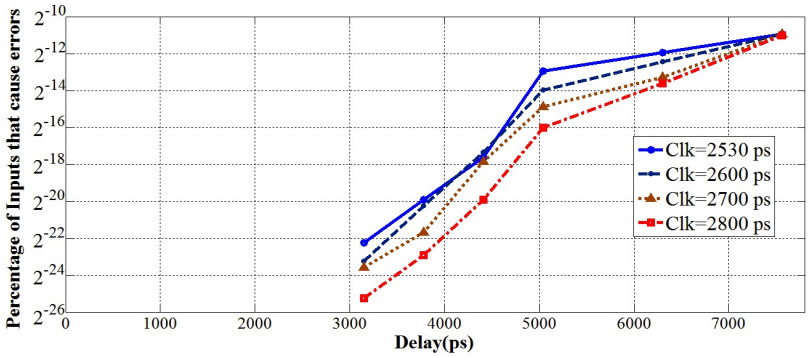


Figure 3.6: Increasing the rare path delay increases the probability of causing an error when random vectors are applied. This delay is allocated to gates according to the delay distribution algorithm. The results are shown for different clock periods.

step of an elliptic curve group operation in OpenSSL could be exploited to recover private ECDH-TLS server keys [21]. Note that while they exploited a software bug as opposed to a hardware bug and a modular reduction as opposed to a multiplication, the attack idea itself is the same as in the original bug attack paper [10].

3.6.1 Fault Model of the Trojan Multiplier

The Trojan Multiplier introduced in the precious Section has a different fault model than the one assumed in [10]. In particular, the output of the Trojan Multiplier does not only depend on the current input but also on the previous inputs, i.e., it has a state. We define the multiplication of two 32-bit numbers a_1, b_1 with our Trojan Multiplier as $\tilde{y} = MUL_{a_0,b_0}(a_1, b_1)$ where a_0, b_0 is the previous input pair to the multiplier. The list F of quadruples (a_0, b_0, a_1, b_1) are all input sequences for which the Trojan Multiplier computes a faulty response:

$$\begin{aligned} &\text{For all } (a_0, b_0, a_1, b_1) \in F : \tilde{y} = MUL_{a_0,b_0}(a_1, b_1) \neq y = a_1 \cdot b_1 \\ &\text{For all } (a_0, b_0, a_1, b_1) \notin F : \tilde{y} = MUL_{a_0,b_0}(a_1, b_1) = y = a_1 \cdot b_1 \end{aligned} \tag{3.3}$$

Outputs computed with the Trojan Multiplier are always represented with a tilde. An ECC scalar multiplication of point $Q \in E$ with an integer k is denoted as $R = k \cdot Q$. An elliptic curve scalar multiplication using the Trojan Multiplier is denoted with an

$\odot$, i.e., $\tilde{R} = k \odot Q$. In the following we assume that an attacker has knowledge of the Trojan Multiplier or access to a chip with the Trojan Multiplier such that the attacker knows for which inputs $\tilde{R} \neq R$.

The attack complexity strongly depends on the probability that a multiplication results in a faulty response. In order to be able to compute this probability we make following definitions:

1. $P_{M(a_1,b_1)}$: Probability that for two random 32-bit integers a_1, b_1 there exits at least one pair of 32-bit integers a_0, b_0 such that $\tilde{y} = MUL_{a_0,b_0}(a_1, b_1)$ computes a faulty response

2. $P_{M(a_1)}$: Probability that for a random 32-bit integers a_1 there exits at least one triplet of 32-bit integers a_0, b_0, b_1 such that $\tilde{y} = MUL_{a_0,b_0}(a_1, b_1)$ computes a faulty response. Probability $P_{M(b_1)}$ is defined in the same fashion.

3. $P_{M(a_0,b_0|a_1,b_1)}$: Probability that for two random 32-bit integers a_0, b_0 and two given integers a_1, b_1 the multiplication $\tilde{y} = MUL_{a_0,b_0}(a_1, b_1)$ computes a faulty response if there exists at least one other input pair a'_0, b'_0 for which $\tilde{y} = MUL_{a'_0,b'_0}(a_1, b_1)$ computes a faulty response

4. $P_{M(a_0|a_1,b_1=b_0)}$: Probability that for a random 32-bit integers a_0, and two given integers a_1, b_1 the multiplication $\tilde{y} = MUL_{a_0,b_0}(a_1, b_1)$ with $b_0 = b_1$ computes a faulty response if there exists at least one other input pair a'_0, b'_0 for which $\tilde{y} = MUL_{a'_0,b'_0}(a_1, b_1)$ computes a faulty response

Furthermore, we make following assumptions regarding these probabilities for the Trojan Multiplier :

1. $P_{M(a_1)} \approx P_{M(b_1)}$ and $P_{M(a_1,b_1)} = P_{M(a_1)} \cdot P_{M(b_1)}$

2. $P_{M(a_0,b_0|a_1,b_1)} \approx 0.09$

3. $P_{M(a_0|a_1,b_1=b_0)} \approx 0.18$

Assumption (1) follows from the fact that both inputs have the same impact on the propagation path of the signal. Hence it is reasonable that both values are equally important to determine if a multiplication fails. Assumption (2) is based on experimental results in which 892 out of 10,000 multiplication failed when a_0 and b_0 are changed randomly while keeping a_1,b_1 constant. Assumption (3) is based on a similar experiment in which 1813 out of 10,000 multiplication failed when a_0 was changed randomly and b_0 was fixed to $b_0 = b_1$ and a_1 was kept constant as well.

3.6.2 Case Study: An ECDH implementation with Montgomery Ladder

For our case study we assume a 255-bit ECDH key agreement with a static public key. Furthermore, we assume the implementation uses the Montgomery Ladder scalar multiplication. The ECDH key agreement works as follows: Given are a standardized public curve E (e.g. Curve25519) and the point $G \in E$. The private key of the server is a 255 bit integer k_s and the corresponding public key is $Q_s = k_s \cdot G$. The key agreement is started by the client by choosing a random 255-bit integer k_c and computing $Q_c = k_c \cdot G$. The client sends Q_c to the server and computes the shared key $R = k_s \cdot Q_s$. The server computes the shared secret key R using Q_c and his secret key k_s by computing $R = k_S \cdot Q_c$. Usually, the key agreement is followed by a handshake to ensure that both the client and the server are now in possession of the same shared session key R.

The general idea of the bug attack is that the attacker makes a key guess of the first l bits of the secret key K_s. Then the attacker searches for a point $Q = m \cdot G$ such that the scalar multiplication $\tilde{R} = k_s \odot Q$ results in a failure if, and only if, the most significant bits of k_s are indeed the l bits the attacker guessed. The attacker then sends Q to the server and completes the ECDH key exchange protocol by making a handshake with the shared key $R = m \cdot Q_s$. If this handshake fails, the expected multiplication error in the Trojan Multiplier has occurred and hence, the attacker knows that his key guess is correct. This way more and more bits of the key are

recovered consecutively. In the Montgomery Ladder scalar multiplication only one bit of the key is processed in each ladder step and the attack works as follows:

1. Input: Elliptic curve E with point $G \in E$ and public server key $Q_s \in E$
2. Initialization: Set $k = 1_{(2)}$
3. Repeat for key bit 2 to 255:
   (a) Define $k_0 = k||0_{(2)}$ [Append a zero to the key k]
   (b) Define $k_1 = k||1_{(2)}$ [Append a one to the key k]
   (c) Repeatedly choose a value m and compute $Q = m \cdot G$ until:
   $(\tilde{P}_i = k_i \odot Q) \neq (P_i = k_i \cdot Q)$ for $i \in \{0, 1\}$
   $(\tilde{P}_j = k_j \odot Q) = (P_j = k_j \cdot Q)$ for $j \neq i$, $j \in \{0, 1\}$
   (d) Send Q to the server and complete handshake with $R = m \cdot Q_s$
   (e) If handshake failed, set $k = k_i$, else set $k = k_j$

The attack described above is a straight forward adaption of the bug attack from [21]. However, in the Trojan multiplier scenario the attack can be improved significantly by adding a precomputation step. The main idea is to not use randomly generated points Q in step 3.c) but to use points Q in which the x-coordinate Q_x contains a b_1 for which the Trojan Multiplier $\tilde{y} = MUL_{a_0,b_0}(a_1, b_1)$ has a high chance to return a faulty response. That is, b_1 is one of the inputs for which the Trojan Multiplier fails. In each step of the Montgomery Ladder algorithm, which is described in subsection 3.6.3, the projective coordinate Z_2 is computed with $Z_2 \leftarrow Z_2 \cdot Q_x$, hence, Q_x, and therefore also b_1, is used in every ladder step. Furthermore, the value Z_2 is different depending on the currently processed key bit. Our improved attack targets this 255-bit integer multiplication $Z_2 \cdot Q_x$ to find a Q such that $(\tilde{P}_i \neq P_i)$ while $(\tilde{P}_j \neq P_j)$ as needed in step 3.c) of the attack algorithm.

Unfortunately, the attacker cannot freely choose Q since the attacker needs to know m such that $Q = m \cdot G$ to finish the handshake. Instead of computing suitable points

for each attack, we propose to search for t suitable points Q during a precomputation step as described below:

1. Input: Elliptic curve E with point $G \in E$
2. Initialization: $m = 1$, $Q = G$
3. Repeat t times:
   (a) $m = m + 1$, $Q = Q + G$
   (b) If Q_x contains b_1, store m and Q in list L

To compute the probability that the 255-bit integer multiplication $Z_2 \cdot Q_x$ fails the used multiplication algorithm is important. We assume that the schoolbook multiplication is used. One 255-bit schoolbook multiplication consists of 64 multiplications of which 8 have b_1 as an operand. Since one of these multiplication is a 31-bit multiplication and we assume that only 32-bit multiplications can trigger the Trojan, 7 32-bit multiplications with b_1 that can trigger the Trojan are performed in each ladder step. Furthermore, due to the FOR loops in the schoolbook multiplication, in 6 of these 7 multiplications $b_0 = b_1$, i.e., the second operand in the multiplication remains unchanged. Note that $P_{M(a_0|a_1,b_1=b_0)} \approx 0.18$ and hence this is actually not a problem but rather helpful. The average number A_Q of points Q that need to be tested until a failure occurs for key bit 1 or 0 is therefore:

$$A_Q = \frac{1}{2} \cdot \frac{1}{P_{M(a_1)} \cdot P_{M(a_0|a_1,b_1=b_0)} \cdot 6 + P_{M(a_1)} \cdot P_{M(a_0,b_0|a_1,b_1)} \cdot 1}$$

Let us assume that the attacker tries to find a point Q for key bit i. Since the attacker searches for a fault in the last Montgomery Ladder step, for every point Q the attacker needs to compute $i - 2$ Montgomery Ladder steps (for the first key bit no step is needed) and then two Montgomery Ladder steps for key bit 1 and 0

respectively to check if the multiplication fails. Hence, in total the attacker needs an average of A_M Montgomery Ladder steps to recover a 255 bit key:

$$A_M = \sum_{i=2}^{255}(i \cdot A_Q) = \frac{255^2 + 255}{2} \cdot A_Q \approx 2^{16} \cdot A_Q$$

To compute t points Q during the precomputation such that b_1 is in Q_x the attacker needs in average $A_P = t \cdot \frac{1}{P_{M(b_1)}}$ point additions. We chose $t = 16 \cdot A_Q$ which results in a failure probability of ca. $3.3 \cdot 10^{-8}$ which should be small enough for all reasonable attack scenarios. Table 3.4 summarizes the attack complexity for our improved bug attack with precomputation for different parameters for the Trojan Multiplier. To put these numbers into perspective, the hardware implementation of curve25519 presented in [75] can compute roughly $2^{39.3}$ Montgomery Ladder steps per second on a Xilinx Zynq 7020 FPGA. Hence, especially for a failure probability of $P_{M(a_1,b_1)} = 2^{-48}$ the attack complexity of 2^{39} Montgomery ladder steps (and 2^{50} point additions that only need to be done once) is quite practical in a real-world scenario. On the other hand, the probability that the Trojan is triggered unintentionally during normal operation is about 2^{-37} which is low enough to not cause problems (see subsection 3.6.3.1 for details).

Table 3.4: Attack complexity of the proposed improved Bug Attack using the Trojan Multiplier assuming a 256 bit curve.

$P_{M(a_1,b_1)}$	2^{-64}	2^{-48}	2^{-32}
Precomputation complexity (point additions)	$2^{66.8}$	$2^{50.8}$	$2^{34.8}$
Storage Requirement	14 PB	55 TB	215 GB
Attack complexity (scalar multiplications)	$2^{30.8}$	$2^{22.8}$	$2^{14.8}$
Attack complexity (Montgomery Ladder Steps)	$2^{46.8}$	$2^{38.8}$	$2^{30.8}$

3.6.3 Montgomery Ladder

To be able to compute the exact attack complexity the details of the Montgomery Ladder are important to determine how many manipulations are performed in each

step. Algorithm 3 and Algorithm 4 describe the details of the assumed Montgomery Ladder implementation.

Algorithm 3: MONTGOMERY LADDER

Input: A 255-bit scalar s and the x-coordinate Q_x of $Q \in E$
Output: c-coordinate P_x of point $P \in E$ with $P = s \cdot Q$

$X_1 \leftarrow 1$; $Z_1 \leftarrow 0$; $X_3 \leftarrow Q_x$; $Z_2 \leftarrow 1$
for $i \leftarrow 254$ *downto 0* **do**
  $b \leftarrow$ bit i of s
  $c \leftarrow$ bit $i-1$ of s for $i < 254$ else $c \leftarrow 0$
  if $b \oplus c = 1$ **then**
    Swap(X_1, X_2)
    Swap(Z_1, Z_2)
  $(X_1, Z_1, X_2, Z_2) \leftarrow LADDERSTEP(Q_x, X_1, Z_1, X_2, Z_2)$
$P_x \leftarrow X_1/Z_1$
return P_x

Algorithm 4: LADDERSTEP OF THE MONTGOMERY LADDER (FOR CURVE 25519)

Input: Q_x, X_1, Z_1, X_2, Z_2
Output: X_1, Z_1, X_2, Z_2

$T_1 \leftarrow X_2 + Z_2$
$X_1 \leftarrow X_2 - Z_2$
$Z_2 \leftarrow X_1 + Z_1$
$X_1 \leftarrow X_1 - Z_1$
$T_1 \leftarrow T_1 \cdot Z_2$
$X_2 \leftarrow X_2 \cdot Z_2$
$Z_2 \leftarrow Z_2 \cdot Z_2$
$X_1 \leftarrow X_1 \cdot X_1$
$T_2 \leftarrow Z_2 - X_1$
$Z_1 \leftarrow T_2 \cdot a24$
$Z_1 \leftarrow Z_1 + X_1$
$Z_1 \leftarrow T_2 \cdot Z_1$
$X_1 \leftarrow Z_2 \cdot X_1$
$Z_2 \leftarrow T_1 - X_2$
$Z_2 \leftarrow Z_2 \cdot Z_2$
$Z_2 \leftarrow Z_2 \cdot Q_x$
$X_2 \leftarrow T_1 + X_2$
$X_2 \leftarrow X_2 \cdot X_2$
return X_1, Z_1, X_2, Z_2

3.6.3.1 Computing the failure probability of a scalar multiplication

In this subsection we describe how the failure probability of a Montgomery Ladder scalar multiplication with schoolbook multiplication on the Trojan Multiplier can be compute. To compute the probability that the computation fails we fist compute the probability that a computation does not fail. As noted previously, in a 255-bit schoolbook integer multiplications with 32-bit word size, 64 multiplications are performed. From this 64 multiplications, 49 multiplications are the multiplications

of two 32-bit numbers, while 6 are 32-bit times 31-bit multiplications and one 31-bit times 31-bit multiplications. We again assume that only 32-bit multiplications can result in a faulty response. In 42 multiplications the second operand is the same as in the previous multiplications and hence the probability that such a multiplication fails is:

$$P_{M(a_1,a_b)} \cdot P_{M(a_0|a_1,b_1=b_0)}$$

For 7 multiplications the failure probability is:

$$P_{M(a_1,a_b)} \cdot P_{M(a_0,b_1|a_1,b_1)}$$

The probability that *no* failure occurs during one Montgomery Ladder step is therefore:

$$(1 - P_{M(a_1,a_b)})^{42} \cdot (1 - P_{M(a_0,b_1|a_1,b_1)})^7$$

A 255-bit scalar multiplication requires 254 Montgomery Ladder steps. Hence the probability that a failure occurs is given by:

$$1 - ((1 - P_{M(a_1,a_b)})^{42} \cdot (1 - P_{M(a_0,b_1|a_1,b_1)})^7)^{254}$$

CHAPTER 4

SIDE-CHANNEL HARDWARE TROJAN FOR PROVABLY-SECURE SCA PROTECTED IMPLEMENTATIONS

[1]In this work, we present a mechanism which shows how easily a stealthy hardware Trojan can be inserted in a provably-secure side-channel analysis protected implementation. Once the Trojan is triggered, the malicious design exhibits exploitable side-channel leakage leading to successful key recovery attacks. Such a Trojan does not add or remove any logic (even a single gate) to the design which makes it very hard to detect. In ASIC platforms, it is indeed inserted by subtle manipulations at the sub-transistor level to modify the parameters of a few transistors. The same is applicable on FPGA applications by changing the routing of particular signals without any resource utilization overhead. The underlying concept is based on a secure masked hardware implementation which does not exhibit any detectable leakage. However, by running the device at a particular clock frequency one of the requirements of the underlying masking scheme is not fulfilled anymore, i.e., the Trojan is triggered, and the device's side-channel leakage can be exploited. We apply our technique to a threshold implementation of the PRESENT block cipher realized in FPGA platform and two different CMOS technologies, and show that triggering the Trojan makes the FPGA and ASICs vulnerable. Although as a case study we show an application of our designed Trojan on the threshold implementation of the PRESENT cipher, our methodology is a general approach and can be applied on any similar circuit.

---

[1]The research presented in this chapter was published in [27] and submitted to [33]. This research is a joint work with Amir Moradi, Thorben Moos, and Maik Ender from Horst Gortz Institute for IT Security, Ruhr Universit at Bochum, Germany.

4.1 Technique

As explained in former section – by means of TI – it is possible to realize hardware cryptographic devices secure against certain SCA attacks. Our goal is to provide a certain situation that an SCA-secure device becomes insecure while it still operates correctly. Such a dynamic transition from secure to insecure should be available and known only to the Trojan attacker. To this end, we target the uniformity property of a secure TI construction. More precisely, we plan to construct a secure and uniform TI design which becomes non-uniform (and hence insecure) at particular environmental conditions. In order to trigger the Trojan (or let say to provide such a particular environmental conditions) for example we select higher clock frequency than the device maximum operation frequency, or lower power supply than the device nominal supply voltage. It should not be forgotten that under such conditions the underlying device should still maintain its correct functionality.

To realize such a scenario – inspired from the stealthy parametric Trojan introduced in Chapter 3 – we intentionally lengthen certain paths of a combinatorial circuit. This is done in such a way that – by increasing the device clock frequency or lowering its supply voltage – such paths become faulty earlier than the other paths. We would achieve our goal if *i*) the faults cancel each others' effect, i.e., the functionality of the design is not altered, and *ii*) the design does not fulfill the uniformity property anymore.

In order to explain our technique – for simplicity without loss of generality – we focus on a 3-share TI construction. As explained in Section 2.3 – ignoring the uniformity – achieving a non-complete shared function $\mathcal{F}^*(.,.,.)$ of a given quadratic function $\mathcal{F}(.)$ is straightforward. Focusing on one output bit of $\mathcal{F}(\boldsymbol{x})$, and representing $\boldsymbol{x}$ by s input bits $\langle x_s, \ldots, x_1 \rangle$, we can write

$$\begin{aligned}\mathcal{F}_i(\langle x_s, \ldots, x_1 \rangle) =& k_0 \oplus k_1 x_1 \oplus k_2 x_2 \oplus \ldots \oplus k_s x_s \oplus \\ & k_{1,2} x_1 x_2 \oplus k_{1,3} x_1 x_3 \oplus \ldots \oplus k_{s-1,s} x_{s-1} x_s.\end{aligned}$$

The coefficients $k_0, \ldots, k_{s-1,s} \in \{0,1\}$ form the Algebraic Normal Form (ANF) of the quadratic function $\mathcal{F}_i : \{0,1\}^s \to \{0,1\}$. By replacing every input bit x_i by the sum of three corresponding shares $x_i^1 \oplus x_i^2 \oplus x_i^3$, the remaining task is just to split the terms in the ANF to three categories in such a way that each category is independent of one share. This can be done by a method denoted by *direct sharing* [17] as

- $\mathcal{F}_i^1(.,.)$ contains the linear terms x_i^2 and the quadratic terms $x_i^2 x_j^2$ and $x_i^2 x_j^3$.
- $\mathcal{F}_i^2(.,.)$ contains the linear terms x_i^3 and the quadratic terms $x_i^3 x_j^3$ and $x_i^3 x_j^1$.
- $\mathcal{F}_i^3(.,.)$ contains the linear terms x_i^1 and the quadratic terms $x_i^1 x_j^1$ and $x_i^1 x_j^2$.

The same is independently applied on each output bit of $\mathcal{F}(.)$ and all three component functions $\mathcal{F}^1(\boldsymbol{x}^2, \boldsymbol{x}^3)$, $\mathcal{F}^2(\boldsymbol{x}^3, \boldsymbol{x}^1)$, $\mathcal{F}^3(\boldsymbol{x}^1, \boldsymbol{x}^2)$ are constructed that fulfill the non-completeness, but nothing about its uniformity can be said.

There are indeed two different ways to obtain a uniform TI construction:

- If s (the underlying function size) is small, i.e., $s \leq 5$, it can be found that $\mathcal{F}(.)$ is affine equivalent to which s-bit class. More precisely, there is a quadratic class $\mathcal{Q}$ which can represent $\mathcal{F}$ as $\mathcal{A}' \circ \mathcal{Q} \circ \mathcal{A}$ (see [18] for an algorithm to find $\mathcal{A}$ and $\mathcal{A}'$ given $\mathcal{F}$ and $\mathcal{Q}$). A classification of such classes for $s = 3$ and $s = 4$ are shown in [17] and for $s = 5$ in [20]. Since the number of existing quadratic classes are restricted, it can exhaustively be searched to find their uniform TI. Note that while for many quadratic classes the direct sharing (explained above) can reach to a uniform TI, for some quadratic classes no uniform TI exists unless the class is represented by a composition of two other quadratic classes [17]. Supposing that $\mathcal{Q}^*(.,.,.)$ is a uniform TI of $\mathcal{Q}(.)$, applying the affine functions $\mathcal{A}'$ and $\mathcal{A}$ accordingly on each input and output of the component function $\mathcal{Q}^*$ would give a uniform TI of $\mathcal{F}(.)$:

$$\mathcal{F}^1(\boldsymbol{x}^2, \boldsymbol{x}^3) = \mathcal{A}' \circ \mathcal{Q}^1\left(\mathcal{A}\left(\boldsymbol{x}^2\right), \mathcal{A}\left(\boldsymbol{x}^3\right)\right),$$
$$\mathcal{F}^2(\boldsymbol{x}^3, \boldsymbol{x}^1) = \mathcal{A}' \circ \mathcal{Q}^2\left(\mathcal{A}\left(\boldsymbol{x}^3\right), \mathcal{A}\left(\boldsymbol{x}^1\right)\right),$$
$$\mathcal{F}^3(\boldsymbol{x}^1, \boldsymbol{x}^2) = \mathcal{A}' \circ \mathcal{Q}^3\left(\mathcal{A}\left(\boldsymbol{x}^1\right), \mathcal{A}\left(\boldsymbol{x}^2\right)\right).$$

This scenario has been followed in several works, e.g., [63, 64, 76, 12, 16].

- Having a non-uniform TI construction, e.g., obtained by direct sharing, we can add *correction terms* to the component functions in such a way that the correctness and non-completeness properties are not altered, but the uniformity may be achieved. For example, the linear terms x_i^2 and/or the quadratic terms $x_i^2 x_j^2$ as correction terms can be added to the same output bit of **both** component functions $\mathcal{F}^1(\boldsymbol{x}^2, \boldsymbol{x}^3)$ and $\mathcal{F}^3(\boldsymbol{x}^1, \boldsymbol{x}^2)$. Addition of any correction term changes the uniformity of the design. Hence, by repeating this process – up to examining all possible correction terms and their combination, which is not feasible for large functions – a uniform construction might be obtained. Such a process has been conducted in [68, 13] to construct uniform TI of PRESENT and Keccak non-linear functions.

  We should here refer to a similar approach called remasking [62, 17] where – instead of correction terms – fresh randomness is added to the output of the component functions to make the outputs uniform. In this case, obviously a certain number of fresh mask bits are required at every clock cycle (see [62, 15]).

Our technique is based on the second scheme explained above. If we make the paths related to the correction terms the longest path, by increasing the clock frequency such paths are the first whose delay are violated. As illustrated, each correction term must be added to two component functions (see Figure 4.1). The paths must be very carefully altered in such a way that the path delay of both instances of the targeted correction term are the longest in the entire design and relatively the same. Hence, at a particular clock frequency both instances of the correction terms are not correctly

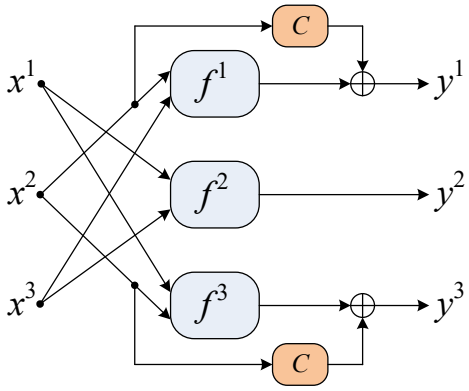


Figure 4.1: Exemplary TI construction with a correction term C.

calculated while all other parts of the design are fault free. This enables the design to still work properly, i.e., it generates correct ciphertext assuming that the underlying design realizes an encryption function. It means that the design operates like an alternative design where no correction terms exists. Hence, the uniformity of the TI construction is not fulfilled and SCA leakage can be exploited. To this end, we should keep a margin between *i*) the path delay of the correction terms and *ii*) the critical path delay of the rest of the circuit, i.e., that of the circuit without correction terms. This margin guarantees that at a certain high clock frequency the correction terms are canceled out but the critical path delay of the remaining circuit is not violated.

We would like to emphasize that in an implementation of a cipher once one of the TI functions generates non-uniform output (by violating the delay of correction terms), the uniformity is not maintained in the next TI functions and it leads to first-order leakage in all further rounds. If the uniformity is achieved by remasking (e.g., in [37]), the above-expressed technique can have the same effect by making the XOR with fresh mask the longest path. Hence, violating its delay in one TI function would make its output non-uniform, but the fresh randomness may make the further rounds of the cipher again uniform.

Based on Figure 4.2, which shows a corresponding timing diagram, the device status can be categorized into four states:

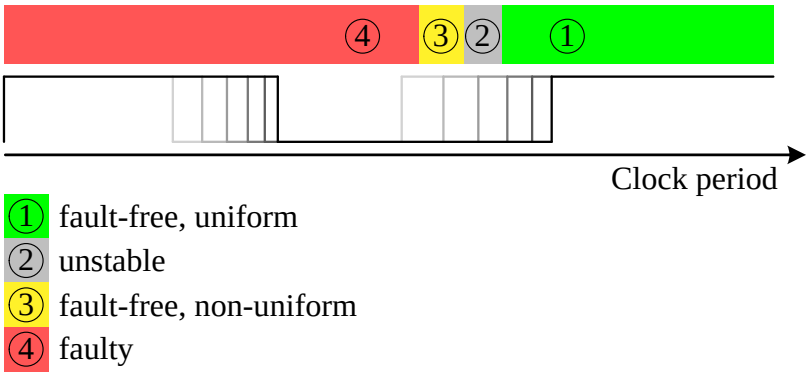


Figure 4.2: Status of the design with Trojan at different clock frequencies.

- at a low clock frequency (denoted by ①) the device operates fault free and maintains the uniformity,
- by increasing the clock frequency (in the ② period), the circuit first starts to become unstable, when indeed the correction terms do not fully cancel each others' effect, and the hold time and/or setup time of the registers are violated,
- by more increasing the clock frequency (in the ③ period), the delay of both instances of the correction term are violated and the circuit operates fault free, but does not maintain the uniformity, and
- by even more increasing the clock frequency (marked by ④) , the clock period becomes smaller than the critical path delay of the rest of the circuit, and the device does not operate correctly.

The aforementioned margin defines the length of the ② period, which is of crucial importance. If it is very wide, the maximum operation frequency of the resulting circuit is obviously reduced, and the likelihood of the inserted Trojan to be detected by an evaluator is increased.

Correct functionality of the circuit is requited to enable the device being operated in the field. Otherwise, the faulty outputs might be detected (e.g., in a communication protocol) and the device may stop operating and prevent collecting SCA traces.

4.2 Application

In order to show an application of our technique, we focus on a first-order TI design of PRESENT cipher [19] as a case study. The PRESENT S-Box is 4-bit cubic bijection $\mathcal{S}$: `C56B90AD3EF84712`. Hence, its first-order TI needs at least $n = 4$ shares. Alternatively, it can be decomposed to two quadratic bijections $\mathcal{S} : \mathcal{F} \circ \mathcal{G}$ enabling the minimum number of shares $n = 3$ at the cost of having extra register between $\mathcal{F}^*$ and $\mathcal{G}^*$ (i.e., TI of $\mathcal{F}$ and $\mathcal{G}$). As shown in [17], $\mathcal{S}$ is affine equivalent to class $\mathcal{C}_{266}$: `0123468A5BCFED97`, which can be decomposed to quadratic bijections with uniform TI. The works reported in [64, 76, 77] have followed this scenario and represented the PRESENT S-Box as $\mathcal{S} : \mathcal{A}'' \circ \mathcal{Q}' \circ \mathcal{A}' \circ \mathcal{Q} \circ \mathcal{A}$, with many possibilities for the affine functions $\mathcal{A}''$, $\mathcal{A}'$, $\mathcal{A}$ and the quadratic classes $\mathcal{Q}'$ and $\mathcal{Q}$ whose uniform TI can be obtained by direct sharing (see Section 4.1).

However, the first TI of PRESENT has been introduced in [68], where the authors have decomposed the S-Box by $\mathcal{G}$: `7E92B04D5CA1836F` and $\mathcal{F}$: `08B7A31C46F9ED52`. They have accordingly provided uniform TI of each of such 4-bit quadratic bijections. We focus on this decomposition, and select $\mathcal{G}$ as the target where our Trojan is implemented. Compared to all other related works, we first try to find a **non-uniform** TI of $\mathcal{G}(.)$, and we later make it uniform by means of correction terms. We start with the ANF of $\mathcal{G}(\langle d, c, b, a\rangle) = \langle g_3, g_2, g_1, g_0\rangle$:

$$g_0 = 1 \oplus a \oplus dc \oplus db \oplus cb, \qquad g_2 = 1 \oplus c \oplus b,$$
$$g_1 = 1 \oplus d \oplus b \oplus ca \oplus ba, \qquad g_3 = c \oplus b \oplus a.$$

One possible sharing of $\boldsymbol{y} = \mathcal{G}(\boldsymbol{x})$ can be represented by $(\boldsymbol{y}^1, \boldsymbol{y}^2, \boldsymbol{y}^3) = (\mathcal{G}^1(\boldsymbol{x}^2, \boldsymbol{x}^3), \mathcal{G}^2(\boldsymbol{x}^3, \boldsymbol{x}^1), \mathcal{G}^3(\boldsymbol{x}^1, \boldsymbol{x}^2))$ as

$$y_0^1 = 1 \oplus a^2 \oplus d^2c^3 \oplus d^3c^2 \oplus d^2b^3 \oplus d^3b^2 \oplus c^2b^3 \oplus c^3b^2 \oplus d^2c^2 \oplus d^2b^2 \oplus c^2b^2,$$
$$y_1^1 = 1 \oplus b^2 \oplus d^3 \oplus c^2a^3 \oplus c^3a^2 \oplus b^2a^3 \oplus b^3a^2 \oplus c^2a^2 \oplus b^2a^2,$$
$$y_2^1 = 1 \oplus c^2 \oplus b^2, \qquad y_3^1 = c^2 \oplus b^2 \oplus a^2,$$

$$y_0^2 = a^3 \oplus d^3c^3 \oplus d^1c^3 \oplus d^3c^1 \oplus d^3b^3 \oplus d^1b^3 \oplus d^3b^1 \oplus c^3b^3 \oplus c^1b^3 \oplus c^3b^1,$$
$$y_1^2 = b^3 \oplus d^1 \oplus c^1a^3 \oplus c^3a^1 \oplus b^1a^3 \oplus b^3a^1 \oplus c^3a^3 \oplus b^3a^3,$$
$$y_2^2 = c^3 \oplus b^3, \qquad y_3^2 = c^3 \oplus b^3 \oplus a^3,$$

$$y_0^3 = a^1 \oplus d^1c^1 \oplus d^1c^2 \oplus d^2c^1 \oplus d^1b^1 \oplus d^1b^2 \oplus d^2b^1 \oplus c^1b^1 \oplus c^1b^2 \oplus c^2b^1,$$
$$y_1^3 = b^1 \oplus d^2 \oplus c^1a^2 \oplus c^2a^1 \oplus b^1a^2 \oplus b^2a^1 \oplus c^1a^1 \oplus b^1a^1,$$
$$y_2^3 = c^1 \oplus b^1, \qquad y_3^3 = c^1 \oplus b^1 \oplus a^1,$$

with $\boldsymbol{x}^{i\in\{1,2,3\}} = \langle d^i, c^i, b^i, a^i \rangle$. This is not a uniform sharing of $\mathcal{G}(.)$, and by searching through possible correction terms we found three correction terms c^1b^1, c^2b^2, and c^3b^3 to be added to the second bit of the above-expressed component functions, that lead us to a uniform TI construction. More precisely, by defining

$$\mathcal{C}^1(\boldsymbol{x}^2, \boldsymbol{x}^3) = c^2b^2 \oplus c^3b^3,$$
$$\mathcal{C}^2(\boldsymbol{x}^3, \boldsymbol{x}^1) = c^1b^1 \oplus c^3b^3,$$
$$\mathcal{C}^3(\boldsymbol{x}^1, \boldsymbol{x}^2) = c^1b^1 \oplus c^2b^2,$$

and adding them respectively to y_1^1, y_1^2, and y_1^3, the resulting TI construction becomes uniform. If any of such correction terms is omitted, the uniformity is not maintained. In the following we focus on a single correction term c^2b^2 which should be added to $\mathcal{G}^1(.,.)$ and $\mathcal{G}^3(.,.)$. A uniform sharing of $\mathcal{F}$ is given in the subsection4.2.1.

4.2.1 Uniform TI of $\mathcal{F}$

Considering $\boldsymbol{y} = \mathcal{F}(\boldsymbol{x})$ and $\boldsymbol{x}^{i\in\{1,2,3\}} = \langle d^i, c^i, b^i, a^i\rangle$ – derived by direct sharing – we present one of its uniform sharing $(\boldsymbol{y}^1, \boldsymbol{y}^2, \boldsymbol{y}^3) = (\mathcal{F}^1(\boldsymbol{x}^2, \boldsymbol{x}^3), \mathcal{F}^2(\boldsymbol{x}^3, \boldsymbol{x}^1), \mathcal{F}^3(\boldsymbol{x}^1, \boldsymbol{x}^2))$ as

$$
\begin{aligned}
y_0^1 &= b^2 \oplus c^2a^2 \oplus c^2a^3 \oplus c^3a^2,\\
y_1^1 &= c^2 \oplus b^2 \oplus d^2a^2 \oplus d^2a^3 \oplus d^3a^2,\\
y_2^1 &= d^2 \oplus b^2a^2 \oplus b^2a^3 \oplus b^3a^2,\\
y_3^1 &= c^2 \oplus b^2 \oplus a^2 \oplus d^2a^2 \oplus d^2a^3 \oplus d^3a^2,
\end{aligned}
$$

$$
\begin{aligned}
y_0^2 &= b^3 \oplus c^3a^3 \oplus c^1a^3 \oplus c^3a^1,\\
y_1^2 &= c^3 \oplus b^3 \oplus d^3a^3 \oplus d^1a^3 \oplus d^3a^1,\\
y_2^2 &= d^3 \oplus b^3a^3 \oplus b^1a^3 \oplus b^3a^1,\\
y_3^2 &= c^3 \oplus b^3 \oplus a^3 \oplus d^3a^3 \oplus d^1a^3 \oplus d^3a^1,
\end{aligned}
$$

$$
\begin{aligned}
y_0^3 &= b^1 \oplus c^1a^1 \oplus c^1a^2 \oplus c^2a^1,\\
y_1^3 &= c^1 \oplus b^1 \oplus d^1a^1 \oplus d^1a^2 \oplus d^2a^1,\\
y_2^3 &= d^1 \oplus b^1a^1 \oplus b^1a^2 \oplus b^2a^1,\\
y_3^3 &= c^1 \oplus b^1 \oplus a^1 \oplus d^1a^1 \oplus d^1a^2 \oplus d^2a^1.
\end{aligned}
$$

4.2.2 Inserting the Trojan

We realize the Trojan functionality by path delay fault model, without modifying the logic circuit. The Trojan is triggered by violating the delay of the combinatorial logic paths that pass through the targeted correction terms c^2b^2. It is indeed a parametric Trojan, which does not require any additional logic. The Trojan is inserted

by modifying a few gates during manufacturing, so that their delay increase and add up to the path delay faults.

Given in Chapter 3 , the underlying method to create a triggerable and stealthy delay-based Trojan consists of two phases: path selection and delay distribution. In the first phase, a set of uniquely-sensitized paths are found that passes through a combinatorial circuit from primary inputs to the primary outputs. Controllability and observability metrics are used to guide the selection of which gates to include in the path to make sure that the path(s) are uniquely sensitized[2]. Furthermore, a SAT-based check is performed to make sure that the path remains sensitizable each time a gate is selected to be added to the path. After a set of uniquely-sensitized paths is selected, the overall delay of the path(s) must be increased so that a delay fault occurs when the path is sensitized. However, any delay added to the gates of the selected path may also cause delay faults on intersecting paths, which would cause undesirable errors and affect the functionality of the circuit. The delay distribution phase addresses this problem by smartly choosing delays for each gate of the selected path to minimize the number of faults caused by intersecting paths. At the same time, the approach ensures that the overall path delay is sufficient for the selected paths to make it faulty.

4.2.2.1 ASIC Platforms

In an ASIC platform, such Trojans are introduce by slightly modifications on the sub-transistor level so that the parameters of a few transistors of the design are changed. To increase the delays of transistors in stealthy ways, there are many possible ways in practice. However, such Trojan is very difficult to be detected by e.g., functional testing, visual inspection, and side-channel profiling, because not a single transistor is removed or added to the design and the changes to the individual

[2]Meaning that the selected paths are the only ones in the circuit whose critical delay can be violated.

gates are minor. Also, full reverse-engineering of the IC would unlikely reveal the presence of the malicious manipulation in the design. Furthermore, this Trojan would not present at higher abstraction levels and hence cannot be detected at those levels, because the actual Trojan is inserted at the sub-transistor level.

A path delay fault in a design is sensitized by a sequence of (at least two) consecutive input vectors on consecutive clock cycles. Its reason is charging/discharging of output capacitances of gates of the path. The delay of each gate is determined by its speed in charging or discharging of its output capacitance. Therefore, if the state of the capacitances of gates (belonging to the targeted path) is not changed (i.e., the capacitances do not charge or discharge), the effect of the path delay fault cannot be propagated along the path. Therefore, to trigger the path delay fault, the consecutive input vectors should change the state of the capacitances of the targeted path.

There are several stealthy ways to change slightly the parameters of transistors of a gate and make it slower in charging/discharging its output capacitance (load capacitance). Exemplary, we list three methods below.

4.2.2.1.1 Decrease the Width Usually a standard cell library has different drive strengths for each logic gate type, which correspond to various transistor widths. Current of a transistor is linearly proportional to the transistor width, therefore a transistor with smaller width is slower to charge its load capacitance. One way to increase the delay of a gate is to substitute it with its weaker version in the library which has smaller width, or to create a custom version of the gate with a narrow width, if the lower level information of the gate is available in the library (e.g., SPICE model). The problem here is that an inspector who test the IC optically, may detect the gate downsizing depending on how much the geometry has been changed.

4.2.2.1.2 Raise the Threshold A common way of increasing delay of a gate is to increase the threshold voltage of its transistors by body biasing or doping manipulation. Using high and low threshold voltages at the same time in a design (i.e., Dual-Vt

design) is very common approach and provides for designer to have more options to satisfy the speed goals of the design. Devices with low threshold voltage are fast and used where delay is critical; devices with high threshold voltage are slow and used where power consumption is important. Body biasing can change the threshold voltage and hence the delay of a gate through changing the voltage between body and source of the transistor [42]. A reverse body bias in which body is at lower voltage than the source, increases the threshold voltage and makes the device slow. In general, transistors with high threshold voltage will response later when an input switches, and conduct less current. Therefore, the load capacitances of the transistors will be charged or discharged more slowly. Dopant manipulation and body biasing, are both very difficult to detect.

4.2.2.1.3 Increase the Gate Length Gate length biasing can increase delay of a gate by reducing the current of its transistors [39]. The likelihood of detection of this kind of manipulation depends on the degree of the modification.

4.2.2.2 FPGA Platforms

In case of the FPGAs, the combinatorial circuits are realized by Look-Up Tables (LUT), in currently-available Xilinx FPGAs, by 6-to-1 or 5-to-2 LUTs and in former generations by 4-to-1 LUTs. The delay of the LUTs cannot be changed by the end users; alternatively we offer the following techniques to make certain paths longer.

4.2.2.2.1 Through Switch Boxes The routings in FPGA devices are made by configuring the switch boxes. Since the switch boxes are made by active components realizing logical switches, a signal which passes through many switch boxes has a longer delay compared to a short signal. Therefore, given a fully placed-and-routed design we can modify the routings by lengthening the selected signals. This is for example feasible by means of `Vivado Design Suite` as a standard tool provided by Xilinx for recent FPGA families and `FPGA Editor` for the older generations. It is in

fact needs a high level of expertise, and cannot be done at HDL level. Interestingly, the resulting circuit would not have any additional resource consumption, i.e., the number of utilized LUTs, FFs and Slices, hence hard to detect particularly if the utilization reports are compared.

4.2.2.2.2 Through route-thrus LUTs Alternatively, the LUTs can be configured as logical buffer. This, which is called *route-thrus*, is a usual technique applied by Xilinx tools to enable routing of non-straightforward routes. Inserting a route-thrus LUT into any path, makes its delay longer. Hence, another feasible way to insert Trojans by delay path fault is to introduce as many as required route-thrus LUTs into the targeted path. It should be noted that the malicious design would have more LUT utilization compared to the original design, and it may increase the chance of being detected by a Trojan inspector. However, none of such extra LUTs realizes a logic, and all of them are seen as *route-thrus* LUTs which are very often (almost in any design) inserted by the FPGA vendor's place-and-route tools. Compared to the previous method, this can be done at HDL level (by hard instantiating route-thrus LUTs).

Focusing on our target, i.e., correction term c^2b^2 in $\mathcal{G}^1(.,.)$ and $\mathcal{G}^3(.,.)$, by applying the above-explained procedure, we found the situation which enables introducing delay path fault into such routes:

- Considering Figure 4.1, the XOR gate which receives the $\mathcal{F}^1$ and $\mathcal{C}$ output should be the last gate in the combinatorial circuit generating y_1^1, i.e., the second bit of $\mathcal{G}^1(.,.)$. The same holds for y_1^3, i.e., the second bit of $\mathcal{G}^3(.,.)$.

- The only paths which should be lengthened are both instances of c^2b^2. Therefore, in case of the FPGA platform we followed both above-explained methods to lengthen such paths, i.e., between *i*) the output of the LUT generating c^2b^2 and *ii*) the input of the aforementioned final XOR gate.

We have easily applied the second method (through route-thrus LUTs) at the HDL level by instantiating a couple of LUTs as buffer between the selected path. More detailed results with respect to the number of required route-thrus LUTs and the achieved frequencies to trigger the Trojan are shown in next Section 5.5.

For the first method (through switch boxes) – since our target platform is a Spartan-6 FPGA – we made use of `FPGA Editor` to manually modify the selected routes. Fig.4.3 shows two routes of a signal with different length.

We should emphasize that this approach is possible if the correction term c^2b^2 is realized by a unique LUT (can be forced at HDL level by hard instantiating or placing such a module in a deeper hierarchy). Otherwise, the logic generating c^2b^2 might be merged with other logic into a LUT, which avoids having a separate path between c^2b^2 and a LUT that realizes the final XOR gate.

4.3 ASIC Implementation

For ASIC platforms, we utilize the stealthy parametric Trojan introduced in Chapter 3. It consists of two main phases: *path selection phase* and *delay distribution phase*. We briefly explain each of these phases in Subsections 4.3.1 and 4.3.2. Our goal is to make the paths related to our target correction term, which is added to two component functions, the longest so that by increasing the clock frequency such paths are the first whose delays are violated. The paths must be very carefully selected and altered in such a way that the path delay of both instances of the targeted correction term are the longest in the entire design and relatively the same. Hence, at a particular clock frequency both instances of the correction terms are not correctly calculated while all other parts of the design are fault free. This enables the design to still work properly.

4.3.1 Rare Path Selection Phase

The path selection phase seeks to find a path π through the netlist of the circuit that passes through the targeted correction term. Note that the delays are not

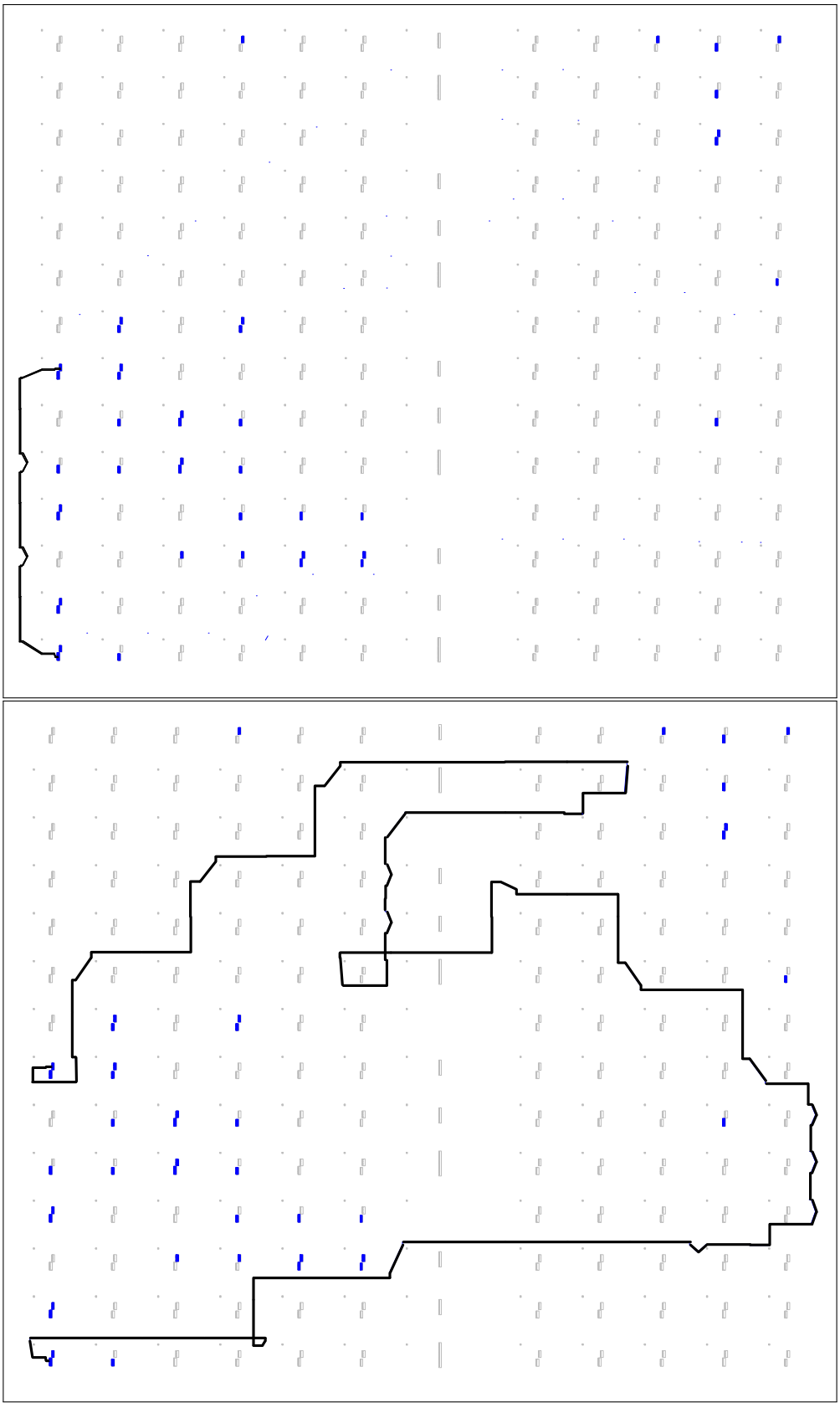


Figure 4.3: Two routes of the same signal in a Spartan-6 FPGA, manually perfromed by `FPGA Editor`.

considered in this phase of the work. Path π is initialized to contain a transition on the targeted correction term node. This initial single-node path π is then extended incrementally backward until reaching the primary inputs, and extended incrementally forward until reaching the primary outputs. The path selection algorithm is given in Alg. 5. Starting from the first transition on the current path π, we repeatedly try to

Algorithm 5: Extracting a hard to trigger sentisizable path passing through a specific node.

Require: A single node π in the netlist of the circuit
Ensure: A sensitizable path π starting at a primary input and ending at a primary output

```
1: while (π does not start at a primary input) do
2:    new_node_candidates = {All transitions that can be prepended to π}
3:    Order new_node_candidates by difficulty of justification.
4:    for (each member n' of new_node_candidates) do
5:       new_subpath π' = prepend n' to the tail of π
6:       if (check-SAT(π')) then
7:          π = π'
8:          Exit for loop.
9:       end if
10:   end for
11: end while
12: while (π does not end at a primary output) do
13:   new_node_candidates = {ALL transitions that can be appended to π}
14:   Order new_node_candidates by difficulty of propagation.
15:   for (each member n' of new_node_candidates) do
16:      new_subpath π' = append n' to the head of π
17:      if (check-SAT(π')) then
18:         π = π'
19:         Exit for loop.
20:      end if
21:   end for
22: end while
```

extend the path back toward the PIs by prepending one new transition to the path. To select such a transition, the algorithm creates a list of candidate transitions that can be prepended to the path, which is sorted according to the difficulty of creating the necessary conditions to justify the transition. Whenever a node is prepended to π to create a candidate path π', the sensitizability of π' is checked by calling *check-SAT* function. In this function SAT-based techniques [26] are used to check sensitizability of the path If the SAT solver returns SAT, then path π' is known to be a subpath of a

sensitizable path from a primary input to a primary output. If this newly added tail node is not a primary input, then the algorithm will again try to extend it backwards.

The forward propagation part is similar to the aforementioned backward propagation, except that it adds nodes to the head of the path until reaching a primary output. At each step of the algorithm, a list of candidates is again formed. In this case, they are ordered according to difficulty of propagation instead of difficulty of justification. Each time a new candidate path is created by adding a candidate node to the existing path, a SAT check is again performed to ensure that the nodes are only added to π if it remains sensitizable.

4.3.2 Delay Distribution Phase

Once paths are selected, the delay of them must be increased so that the total path delays exceed the clock period and errors occur when the paths are sensitized. Choosing where to add delay on the paths must be done carefully, because the gates along the chosen paths are also part of many other intersecting or overlapping paths. Any delay added to the chosen paths therefore may cause errors even when the chosen paths are not sensitized. Genetic algorithm is used to smartly decide the delay of each gate along with some constraints to restrict the allowed solution space, and a fitness function for evaluating solutions.

Total Path Delay Constraint: Assume each of the chosen paths π includes n gates and target path delay is D. This constraint specifies that the sum of assigned delays along the path is equal to the target path delay D. To cause an error, D must exceed the period ④.

$$D = \sum_{i=0}^{n} d_i \tag{4.1}$$

Gate Delay Constraint: Assume d_i' represents the nominal delay of the ith gate on the chosen path π, and s_i represents the slack metric associated with the same gate. Each slack parameter s_i describes how much delay can be added to the corresponding gate without causing the path to exceed the period ④. The slack for each gate is

computed as a function of the nominal delay of the gate, data dependency, and the clock period [29, 82]. Because the targeted path delay D does exceed the period ④, gate delays are allowed to exceed their computed slack. The following equation shows this constraint where c is a constant.

$$d'_i + s_i - c \leq d_i \leq d'_i + s_i + c \tag{4.2}$$

Fitness Function: The cost function consists two parts; *i*) the faults cancel each others' effect, i.e., the faults on targeted correction term in two functions $\mathcal{G}^1$ and $\mathcal{G}^3$ are happen at the same time and cancel the effect of each others so the functionality of the design is not altered, and *ii*) the design does not fulfill the uniformity property anymore. To cover both cases in our final cost function we define it as the following equation in which first term corresponds to case (i) and the second term corresponds to case (ii). Our goal is to minimize this cost function.

$$\text{Cost}_F(d_1, ..., d_n) = \text{ErrorRate}_{\text{design}} + 1/\text{ErrorRate}_{\mathcal{G}^1 \text{ and } \mathcal{G}^3} \tag{4.3}$$

We use random simulation to evaluate the cost of any delay assignment. When the genetic algorithm in Matlab [1] needs to evaluate the cost of a particular delay assignment, it does so by executing a timing simulator. The timing simulator, in our case ModelSim, applies test vectors to the circuit-under-evaluation and a golden copy of the circuit and compares the respective outputs to count the number of errors.

4.4 FPGA Practical Results

4.4.1 Design Architecture

We made use of the above-explained malicious PRESENT TI S-Box in a design with full encryption functionality. The underlying design is similar to the *Profile 2* of [68], where only one instance of the S-Box is implemented. The nibbles are serially

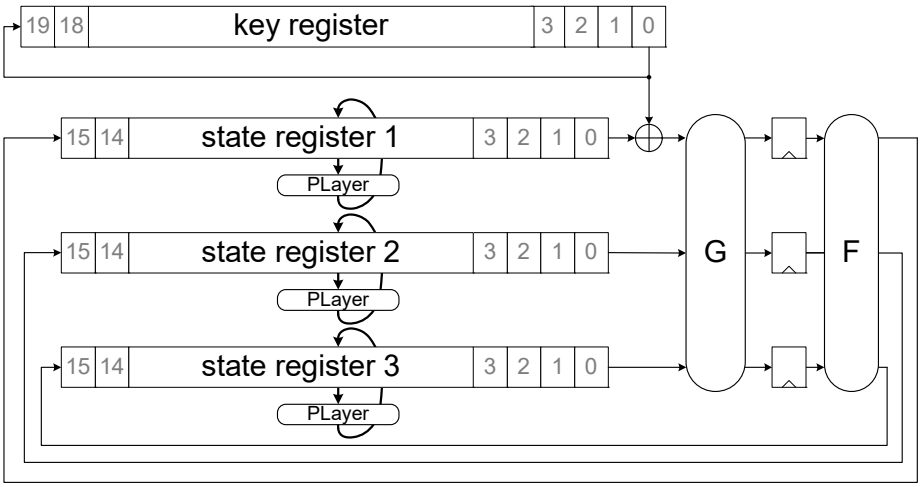


Figure 4.4: Design architecture of the PRESENT TI as the case study.

shifted through the state register as well as through the S-Box module while the `PLayer` is performed in parallel in one clock cycle. Following its underlying first-order TI, the 64-bit plaintext is provided by three shares, i.e., second-order Boolean masking, while the 80-bit key is not shared (similar to that of [68] and [15]). Figure 4.4 shows an overview of the design architecture, which needs 527 clock cycles for a full encryption after the plaintext and key are serially shifted into the state (resp. key) registers.

We should here emphasize that the underlying TI construction is a first-order masking, which can provably provide security against first-order SCA attacks. However, higher-order attacks are expected to exploit the leakage, but they are sensitive to noise [70] since accurately estimating higher-order statistical moments needs huge amount of samples compared to lower-order moments. It is indeed widely known that such masking schemes should be combined with hiding techniques (to reduce the SNR) to practically harden (hopefully disable) the higher-order attacks. As an example we can refer to [64], where a TI construction is implemented by a power-equalization technique. We instead integrated a noise generator module into our target FPGA to increase the noise and hence decrease the SNR. The details of the integrated noise generator module is given in subsection4.4.2. Note that without such a noise generator module, our design would be vulnerable to higher-order attacks and no

Trojan would be required to reveal the secret. Therefore, the existence of such a hiding countermeasure to make higher-order attacks practically hard is essential.

The design is implemented on a Spartan-6 FPGA board SAKURA-G, as a platform for SCA evaluations [2]. In order to supply the PRESENT core with a high clock frequency, a Digital Clock Manager (DCM) has been instantiated in the target FPGA to multiply the incoming clock by a factor of 8. The external clock was provided by a laboratory adjustable signal generator to enable evaluating the design under different high clock frequencies.

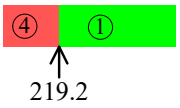
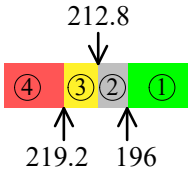
Table 4.1 shows the resource utilization (excluding the noise generator) as well as the achieved margins for the clock frequency considering *i*) the original design, *ii*) malicious design made by *through switch boxes* method and *iii*) malicious design made by *through route-thrus LUTs* technique. It is noticeable that the first malicious design does not change the utilization figures at all since lengthening the routes are done only through the switch boxes (see Fig.4.3). Using the second method – in order to achieve the same frequency margins – we added 4 route-thru LUTs (at the HDL level) to each path of the targeted correction term. This led to 8 extra LUT utilization and 4 more Slices; we would like to mention that the combinatorial circuit of the entire TI S-Box (both $\mathcal{G}^*$ $\mathcal{F}^*$) would fit into 29 LUTs (excluding the route-thru ones).

Regarding the frequency ranges, shown in Table 4.1, it can be seen that the maximum clock frequency of the malicious design is decreased from 219.2 MHz to 196 MHz, i.e., around 10% reduction. However, both ② and ③ periods are very narrow, that makes it hard to be detected either by a Trojan inspector or by an SCA evaluator.

4.4.2 Noise Generator

We have built a noise generator as an independent module, i.e., it does not have any connection to the target PRESENT design and operates independently. We followed one the concepts introduced in [38]. As shown by Figure 4.5, it is made as

Table 4.1: Performance figure of our PRESENT-80 encryption designs.

Design	Method	Utilization				Frequency [MHz]
		FF	LUT		Slice	
			logic	*route-thrus*		
Original	-	299	291	35	226	
Malicious	*switch box*	299	291	35	226	
Malicious	*route-thru LUT*	299	291	43	230	

a combination of a ring oscillator, an LFSR, and several shift registers. The actual power is consumed by the shift registers. Every shift register instantiates a `SRLC32E` primitive, which is a 32-bit shift register within a single LUT inside a `SLICEM`. The shift registers are initialized with the consecutive values of 01. Every shift register's output is feedback to its input and shifted by one at every clock cycle when enabled. Thus, every shift operation toggles the entire bits inside the registers, which maximizes the power consumption of the shift register.

The ring oscillator, made of 31 inverter LUTs, acts as the clock source inside the noise module for both the LFSR and the shift registers. The LFSR realizes the irreducible polynomial $x^{19} + x^{18} + x^{17} + x^{14} + 1$ to generate a pseudo-random clock enable signal for the shift registers.

We instantiated 4×8 instances of the shift register LUTs, fitting into 8 Slices. The ring oscillator required 17 Slices (as stated, made of 31 inverters), and the LFSR fits into 2 Slices, made by 1 LUT for the feedback function, 2 FFs and 2 shift register LUTs. Overall, the entire independent noise generator module required 27 Slices.

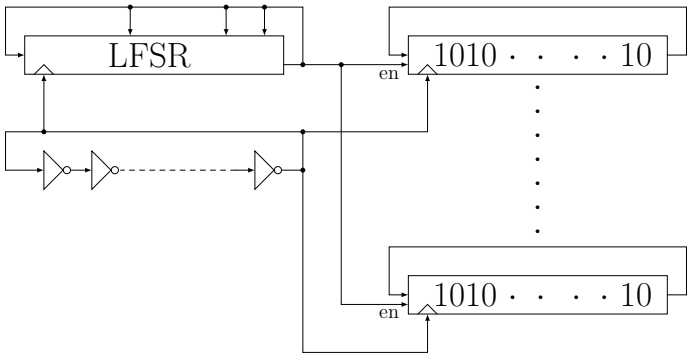


Figure 4.5: Block diagram of the noise generator.

4.4.3 SCA Evaluations

4.4.3.1 Measurement Setup.

For SCA evaluations we collected power consumption traces (at the Vdd path) of the target FPGA by means of a digital oscilloscope at sampling rate of 1GS/s. It might be thought that when the target design runs at a high frequency $> 150\,\text{MHz}$, such a sampling rate does not suffice to capture all leaking information. However, power consumption traces are already filtered due to the PCB, shunt resistor, measurement setup, etc. Hence, higher sampling rate for such a setting does not improve the attack efficiency[3], and often the bandwidth of the oscilloscope is even manually limited for noise reduction purposes (see [69]).

4.4.3.2 Methodology.

In order to examine the SCA resistance of our design(s) in both settings, i.e., whether the inserted Trojan is triggered or not, we conducted two evaluation schemes. We first performed non-specific t-test (fixed versus random) [78, 34] to examine the existence of detectable leakage. Later in case where the Trojan is triggered, we also conduct key-recovery attacks.

[3]It is not the case for EM-based analyses.

It should be mentioned that both of our malicious designs (see Table 4.1) operate similarly. It means that when the Trojan is triggered, the evaluation of both designs led to the same results. Therefore, below we exemplary show the result of the one formed by *through route-thrus LUTs*.

To validate the setup, we start with a non-specific t-test when the PRNG of the target design (used to share the plaintext for the TI PRESENT encryption) is turned off, i.e., generating always zero instead of random numbers. To this end, we collected 100,000 traces when the design is operated at 168 MHz, i.e., the Trojan is not triggered. We followed the concept given in [78] for the collection of traces belonging to fixed and random inputs. The result of the t-test (up to third-order) is shown in Figure 4.6, confirming the validity of the setup and the developed evaluation tools.

To repeat the same process when the PRNG is turned on, i.e., the masks for initial sharing of the plaintext are uniformly distributed, we collected 100,000,000 traces for non-specific t-test evaluations. In this case, the device still operates at 168 MHz, i.e., the Trojan is not triggered. The corresponding results are shown in Figure 4.7. Although the underlying design is a realization of a first-order secure TI, it can be seen from the presented results that second- and third-order leakages are also not detectable. As stated before, this is due to the integration of the noise generator module which affects the detectability of higher-order leakages (see subsection4.4.2).

As the last step, the same scenario is repeated when the clock frequency is increased to 216 MHz, where the design is in the ③ period, i.e., with correct functionality and without uniformity. Similar to the previous experiment, we collected 100,000,000 traces for a non-specific t-test, whose results are shown in Figure 4.8. As shown by the graphics, there is detectable leakage through all statistical moments but with lower t-statistics compared to the case with PRNG off. Therefore, we have also examine the feasibility of key recovery attacks. To this end, we made use of those collected traces which are associated with random inputs, i.e., around 50,000,000 traces of the last non-specific t-test. We conducted several different CPA and DPA attacks

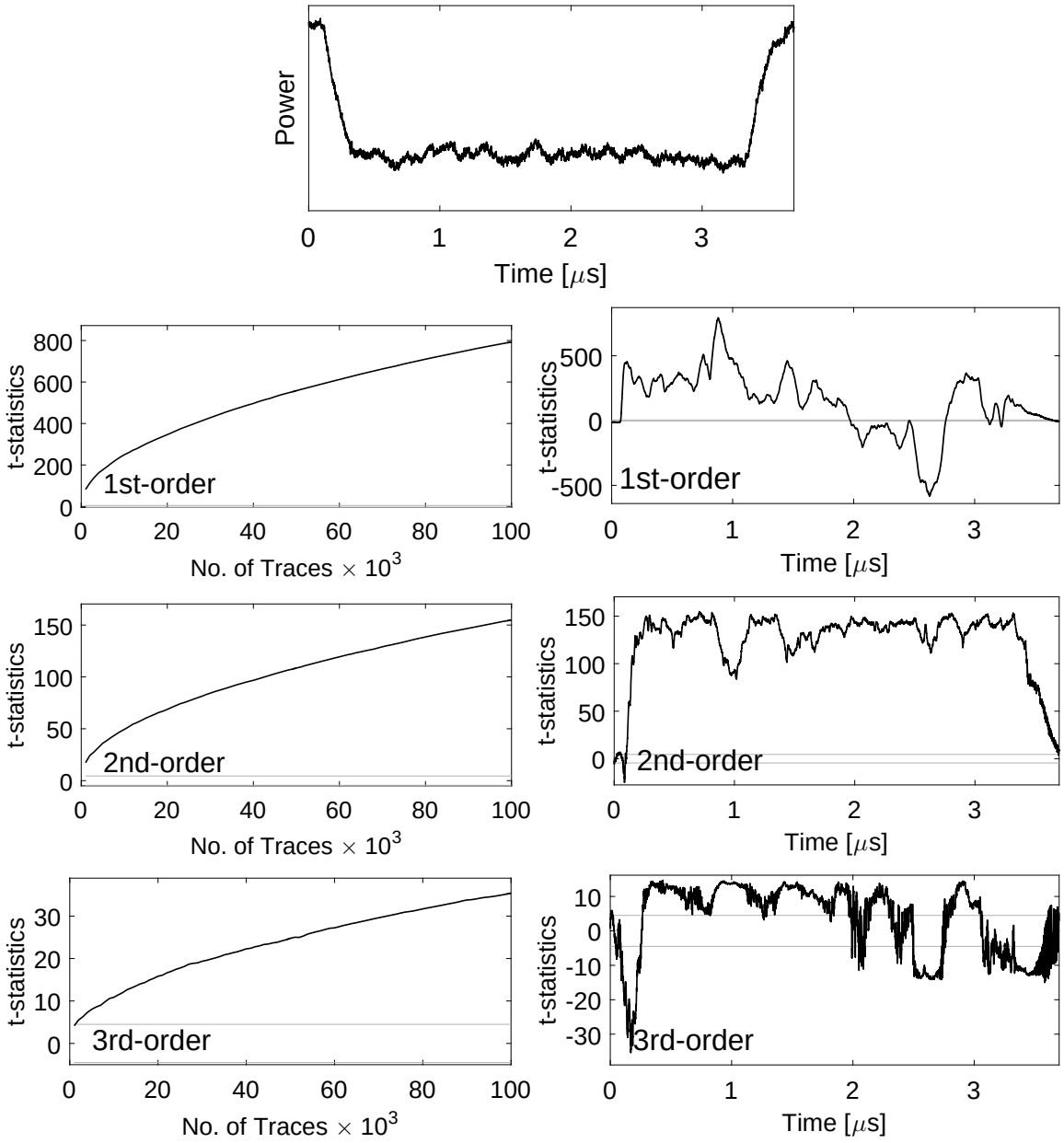


Figure 4.6: PRNG off, clock 168 MHz (Trojan not triggered), (top) a sample power trace, t-test results (right) with 100,000 traces, (left) absolute maximum over the number of traces.

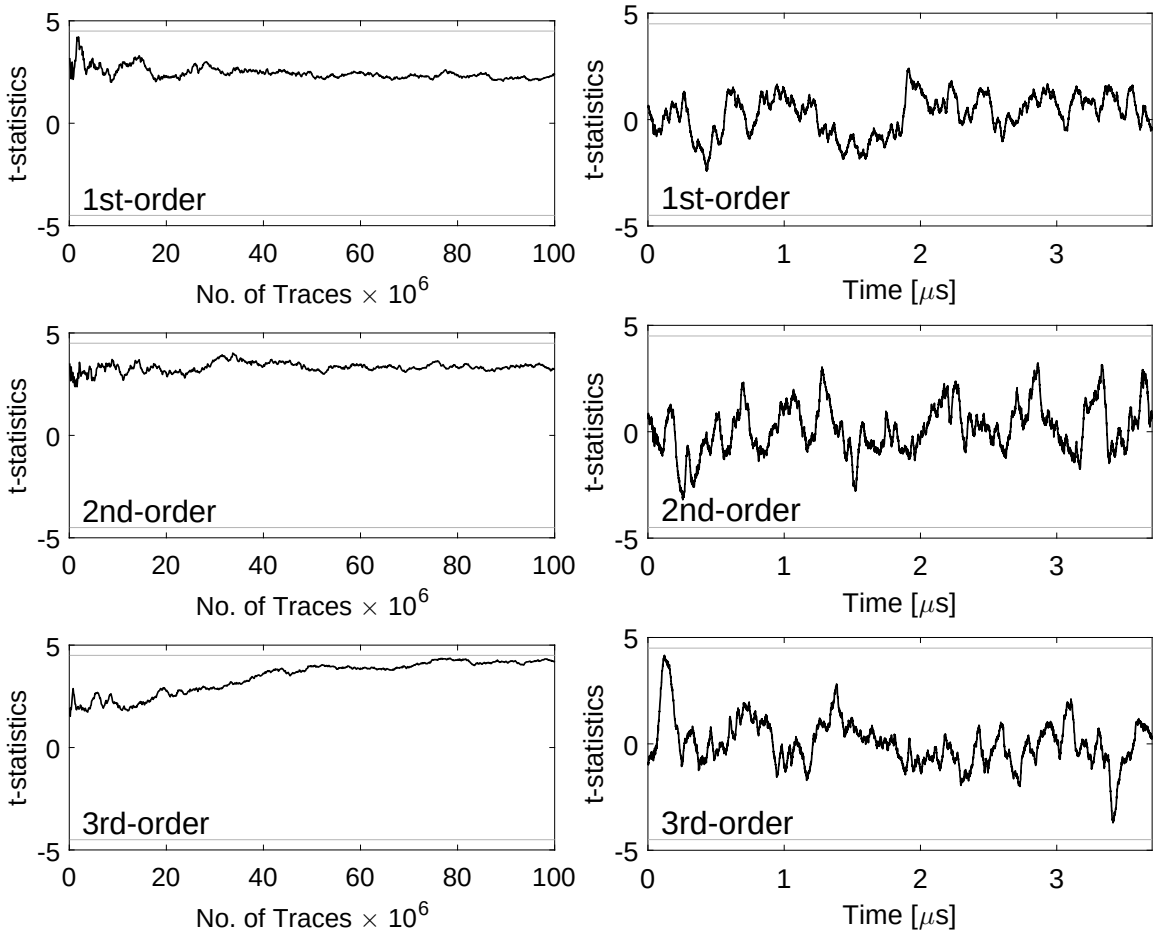


Figure 4.7: PRNG on, clock 168 MHz (Trojan not triggered), t-test results (right) with 100,000,000 traces, (left) absolute maximum over the number of traces.

considering intermediate values of the underlying PRESENT encryption function. The most successful attack was recognized as classical DPA attacks [48] targeting a key nibble by predicting an S-Box output bit at the first round of the encryption. As an example, Figure 4.9 presents an exemplary corresponding result.

4.5 ASIC Practical Results

In this section we describe how we have designed and implemented first ASIC prototypes incorporating such a malicious design. We then verify that the resulting chips are indeed resistant against side-channel attacks when the Trojan is not triggered and that this resistance can be nullified when triggering it.

Section 4.4 demonstrated by practical experiments that the proposed hardware Trojan and the presented implementation techniques are valid on FPGA-based platforms. Here, we aim to provide a similar case study, but with respect to ASIC platforms. In this regard we carried out the described design stages and implemented the trojanized PRESENT threshold implementation circuit in two different process technologies,90 nm and 65 nm low power CMOS. Both ASICs, which can be seen in Figure 4.10, were developed using an identical design procedure, including the usage of low, high and standard threshold voltage cells, and were manufactured by the same foundry.

The size of both chips is 2mm x 2mm. The side-channel resistant PRESENT TI cores containing the parametric SCA Trojans have been placed and routed in clearly delimited rectangular areas, which are marked in red color with a white cross in both layout schematics 4.10a and 4.10b, taken from the *Synopsys IC Compiler (Version 2016.12)* software.

We made use of the malicious PRESENT TI S-Box that has been introduced in the previous sections and embedded it in a design with full encryption functionality shown in Figure 4.4.

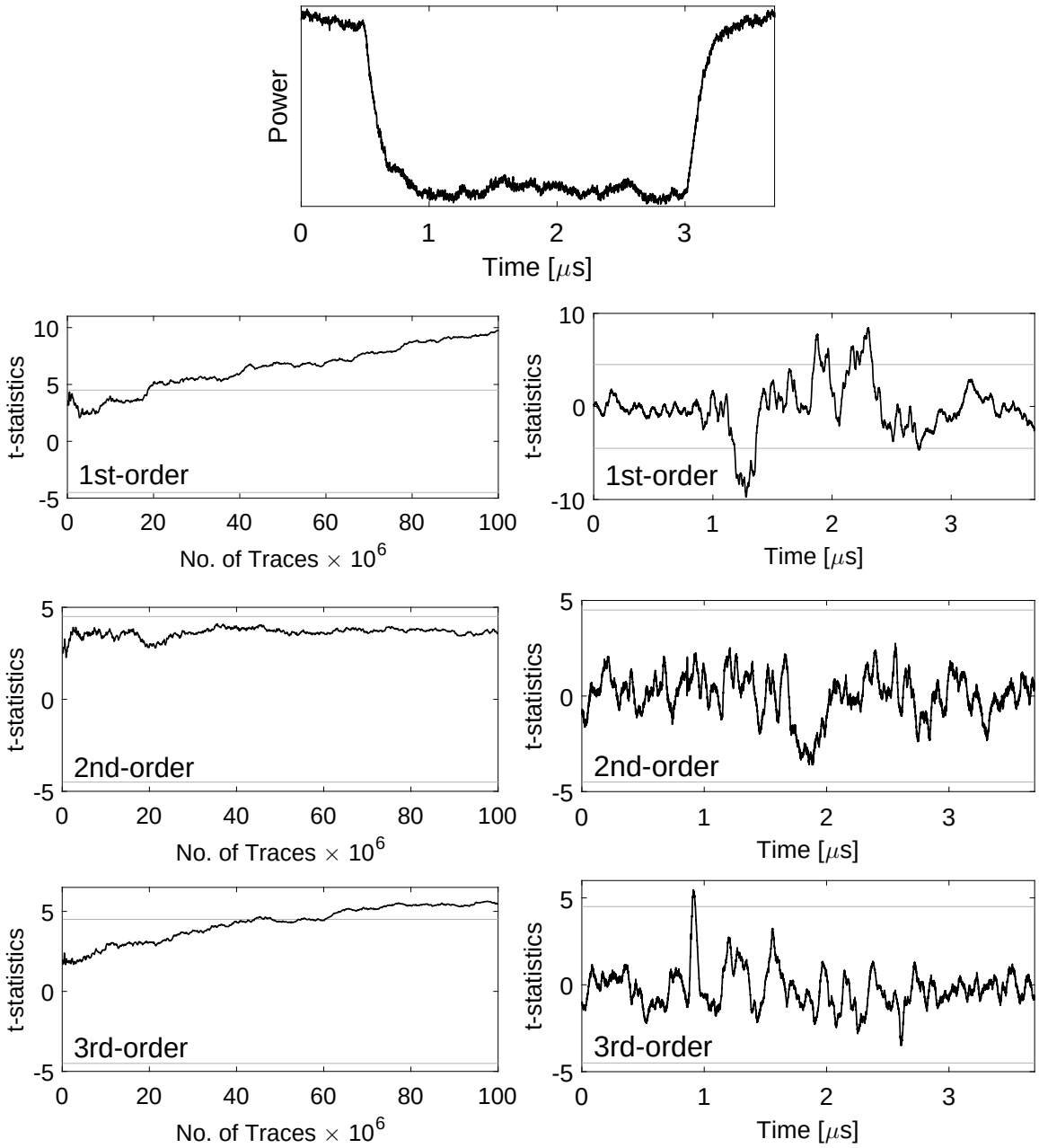


Figure 4.8: PRNG on, clock 216 MHz (Trojan triggered), (top) a sample power trace, t-test results (right) with 100,000,000 traces, (left) absolute maximum over the number of traces.

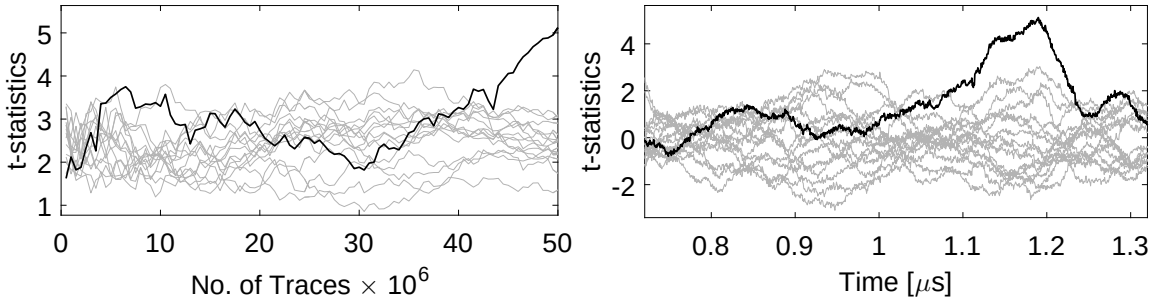
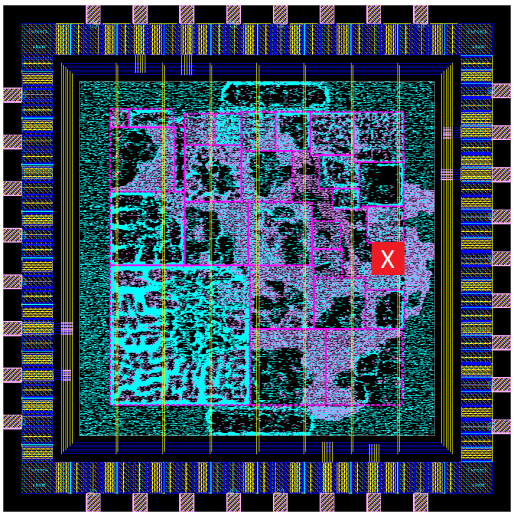
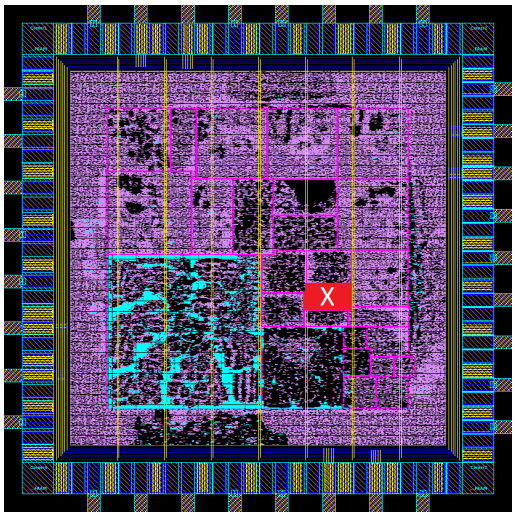


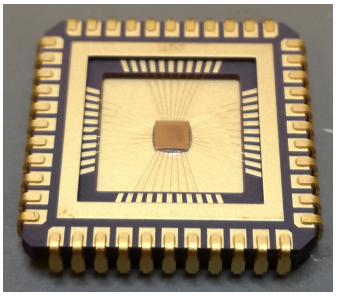
Figure 4.9: PRNG on, clock 216 MHz (Trojan triggered), 50,000,000 traces, DPA attack result targeting a key nibble based on an S-Box output bit at the first round.



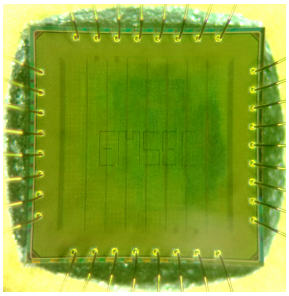
(a) Layout schematic 65 nm ASIC



(b) Layout schematic90 nm ASIC



(c) Photo of packaged 90 nm ASIC



(d) Microscope photo 90 nm ASIC

Figure 4.10: Layout schematic and photos of the ASIC prototypes.

Synthesizing the unaltered threshold implementation of the PRESENT S-Box (i.e., without the inserted Trojan) in the 90 nm and 65 nm target libraries revealed that the design could potentially meet clock frequencies in the GHz range, even when operated under worst case operating conditions (i.e., low supply voltage, high temperature). Unfortunately, no digital IO cells were available in our target technologies that could reliably propagate such a high-frequency clock into the circuit. Thus, when inserting the Trojan in the proposed way, i.e., by subtle manipulations at the sub-transistor level, and keeping period ③ small and stealthy, it could never be triggered, due to the restrictions of the IO cells and the extremely high performance of the circuit in the target technologies.

This observation already shows that implementing and testing such a design on an ASIC is more challenging than on an FPGA, due to the much higher performance of ASICs. In this regard we have to conclude that an ultra-lightweight block cipher implementation like the serialized PRESENT, implemented in an advanced CMOS technology with small propagation delays, may not be the optimal choice for integrating such a Trojan on an ASIC in the most stealthy way. Yet, to keep the results comparable to those in [27], we stick to this example and find a workaround for the IO restriction.

Another difficulty when developing ASIC prototypes is the extensive amount of time and monetary resources that have to be invested. Thus, it is desirable to obtain a fully functioning prototype in the first attempt when designing a test chip. However, this is particularly difficult to achieve when the functionality of the design depends highly on the exact timing of certain signal paths in such a way that even small deviations from the predicted behavior can invalidate crucial assumptions. In such a case the designer has to trust its foundry that the characterized timing information included in the standard cell libraries and simulation models perfectly reflects the reality – which is hardly ever possible due to process variations. Thus, even commercial IC design houses often require multiple generations of prototypes that need to be characterized and adapted between each iteration to finally end up with a marketable

end-user product. Unlike FPGA platforms where a new HDL design can be synthesized and implemented within a few minutes and without any additional cost, which allows for trial & error approaches, an IC implementation requires at least several months per tape out as well as a significant amount of money, even when sharing a wafer between multiple projects. Thus, for our case study, in order to not require multiple IC manufacturing iterations, but rather obtain a working prototype in the first attempt, we chose to limit the potential sources of error at the cost of sacrificing a part of the potential stealthiness of the Trojan. In particular, we chose to realize the delay which needs to be distributed among the selected paths partially by so-called delay gates[4] and optimize for a broad frequency range that triggers the Trojan while the PRESENT core still encrypts correctly (i.e., period ③). A delay gate does not have any logical functionality but simply propagates its input signal with a certain propagation delay to its output. Clearly, inserting delay gates into the masked S-Box makes the Trojan less stealthy than sub-transistor level modifications. The same is true for a significant reduction of the overall operating frequency of the circuit as it can be observed in the results presented in the following (this reduction is neccessary due to the restrictions of the IO cells). However, we would like to stress that this case study is simply proving the conceptual soundness of the approach, in the sense that inserting this delay-based Trojan makes a side-channel resistant implementation vulnerable when increasing the clock frequency beyond a certain point. It is planned to demonstrate the stealthiness of the Trojan on ASIC platforms in further case studies. In many cases, for example targeting more complex non-linear functions (like the AES S-Box) or less advanced CMOS technologies (implying larger delays), such a use of additional delay gates will not be required since the critical path of the design actually restricts the maximum operating frequency of the design (and not the limitations of the IO cells). Again, we chose the PRESENT threshold implementation as a case study here to keep the

[4]Those gates were required since selecting even the slowest cells (high threshold voltage, low drive strength) could not add enough delay in order to make the Trojan triggerable through the IO cells.

Table 4.2: Area comparison (post-layout) of PRESENT TI implementation with and without inserted Trojan (realized by delay gates).

Technology node	**Area w/o Trojan**	**Area w/ Trojan**	**Overhead**
65 nm	4988.5 GE	5006.5 GE	+0.36%
90 nm	4807.8 GE	4825.8 GE	+0.37%

Table 4.3: Frequency ranges for the different design states.

Status	**65 nm ASIC**	**90 nm ASIC**
①	$f \leq 33$ MHz	$f \leq 56$ MHz
②	33 MHz $< f \leq 38$ MHz	56 MHz $< f \leq 61$ MHz
③	38 MHz $< f \leq$ appr. 1 GHz	61 MHz $< f \leq$ appr. 1 GHz
④	appr. 1 GHz $< f$	appr. 1 GHz $< f$

results comparable to [27]. And even in our case, where we particularly aimed for a broad range of period ③, the overhead in terms of area is very small, even less than half a percent as apparent from Table 4.2. The range of clock frequencies that cause a certain state of the trojanized design can be seen in Table 4.3. As described before, state ③ has the broadest frequency range and can easily be targeted by setting the clock frequency above 38 MHz for the 65 nm ASIC and 56 MHz for the 90 nm ASIC. The upper limit where the output of the circuit becomes faulty is an approximation, since it could not be determined experimentally due to the limitation of the IO cells.

4.5.1 Measurement Setup

In order to perform the SCA evaluations on the ASIC prototypes we built a simple custom measurement board. Since the ASICs have been packaged in JLCC-44 package (see Figure 4.10c), the custom board provides a corresponding PLCC-44 socket as well as connectors for a BASYS-3 FPGA board (containing an Artix-7 FPGA) to control the communication between PC and the ASIC. We measured the power consumption of the ASICs in the V_{dd} path by means of a digital sampling oscilloscope at a fixed sampling rate of 200 samples per clock cycle. Since the operating frequency varies

between the different scenarios (Trojan triggered or not triggered), fixing the number of samples per clock cycle (instead of per time period) is the most fair evaluation method.

4.5.2 SCA Results

We evaluate the SCA resistance of our designs in three different settings using a non-specific t-test (fixed versus random) [34, 78] to examine the existence of detectable leakage. First, to validate the correct functionality of the setup, we start with a non-specific t-test when the PRNG of the target design (used to share the plaintext for the TI PRESENT encryption) is turned off, i.e., generating always zero instead of random numbers. Afterwards, we activate the PRNG and operate the design at low frequency in order to not activate the Trojan. Then, when the PRNG is still running we increase the clock frequency in order to activate the Trojan. In the latter case we also conduct key-recovery attacks.

4.5.2.1 Results on 90 nm ASIC

We first collected 1,000,000 traces with PRNG switched off when the design is operated at 25 MHz, i.e., the Trojan is not triggered. We followed the concept given in [78] for the collection of traces belonging to fixed and random inputs. Figure 4.11 shows the corresponding t-test results.

As expected a significant amount of detectable leakage can be observed in all moments, confirming the validity of the setup and the developed evaluation tools.

To repeat the same process when the PRNG is turned on, i.e., the masks for initial sharing of the plaintext are randomly chosen and uniformly distributed, we collected 50,000,000 traces for non-specific t-test evaluations. In this case, the device still operates at 25 MHz, i.e., the Trojan is not triggered. The corresponding results are shown in Figure 4.12.

It can be seen that no leakage is detected in any of the three statistical moments after 50,000,000 traces. However, when observing the progress of the maximum

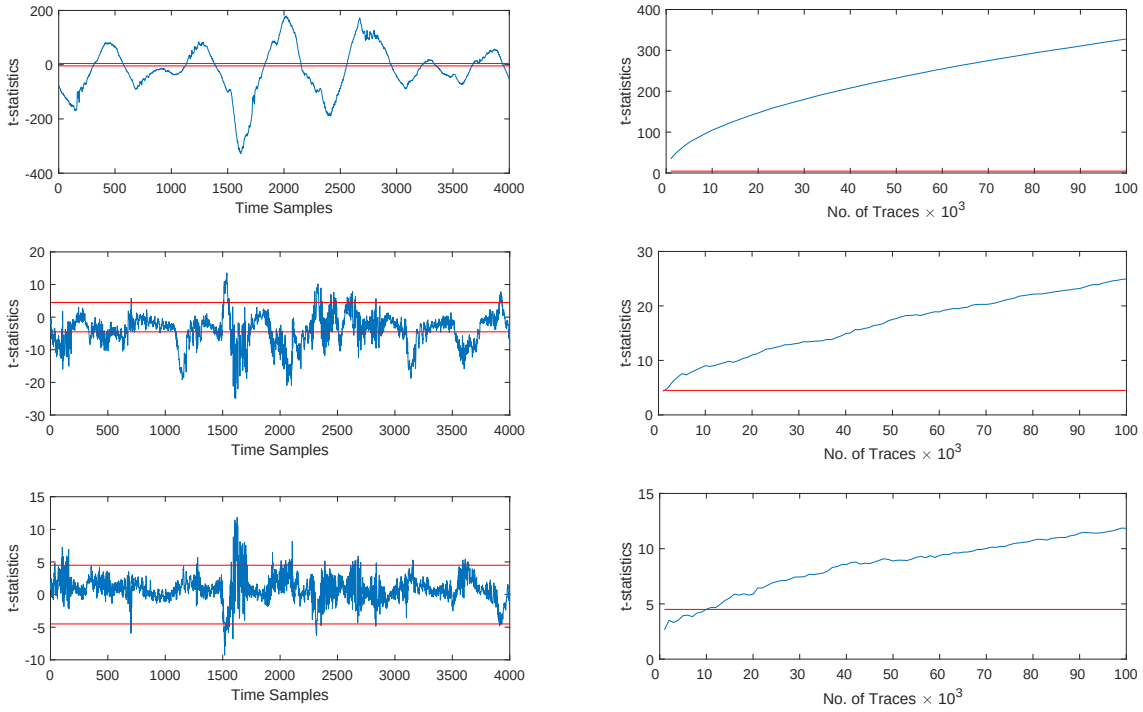


Figure 4.11: 90 nm ASIC, PRNG off, clock frequency 25 MHz (trojan not triggered), t-test results with 1 million traces (left), absolute maximum t-value over the number of traces (right).

absolute t-value in the second-order moment over the number of traces one may notice that the 4.5 threshold is occasionally exceeded. We should emphasize here that the underlying TI construction is a first-order masking, which can provide provable security against first-order SCA attacks. However, higher-order attacks (in this case second-order attacks already) are expected to exploit the leakage, but they are sensitive to the noise level [70] since accurately estimating higher-order statistical moments requires huge amounts of samples compared to lower-order moments. Thus, the second-order leakage is not unexpected, but the noise level seems too large to reliably detect (or exploit) this leakage.

As the last step, the same scenario is repeated when the clock frequency is increased to 85 MHz, where the design is in the ③ period, i.e., with correct functionality and without uniformity. Similar to the previous experiment, we collected 50,000,000 traces for a non-specific t-test, whose results are shown in Figure 4.13.

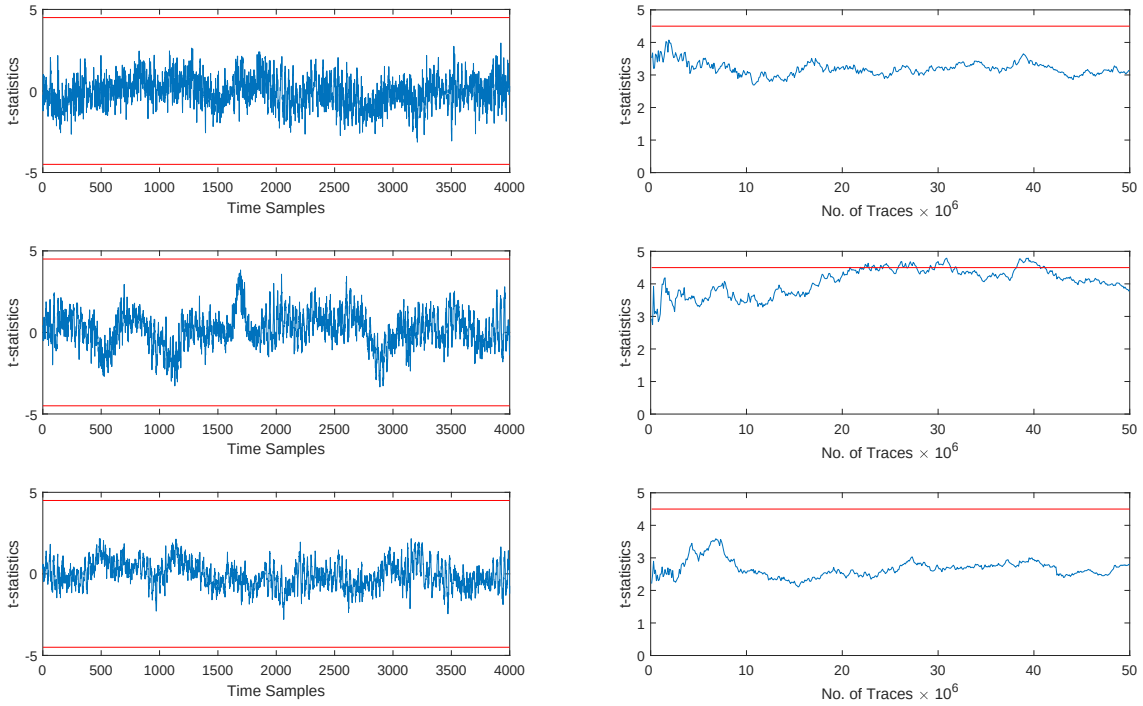


Figure 4.12: 90 nm ASIC, PRNG on, clock frequency 25 MHz (trojan not triggered), t-test results with 50 million traces (left), absolute maximum t-value over the number of traces (right).

As shown by the graphics, there is detectable leakage through the first and second statistical moment but with lower t-statistics compared to the case with PRNG off. Therefore, we also have to examine the feasibility of key recovery attacks. To this end, we made use of those collected traces which are associated with random inputs, i.e., around 25,000,000 traces of the last non-specific t-test. We conducted several different CPA and DPA attacks considering intermediate values of the underlying PRESENT encryption function. The most successful attack was recognized as classical DPA attack [48] targeting a key nibble by predicting an S-Box output bit at the first round of the encryption. As an example, Figure 4.14 presents an exemplary corresponding result.

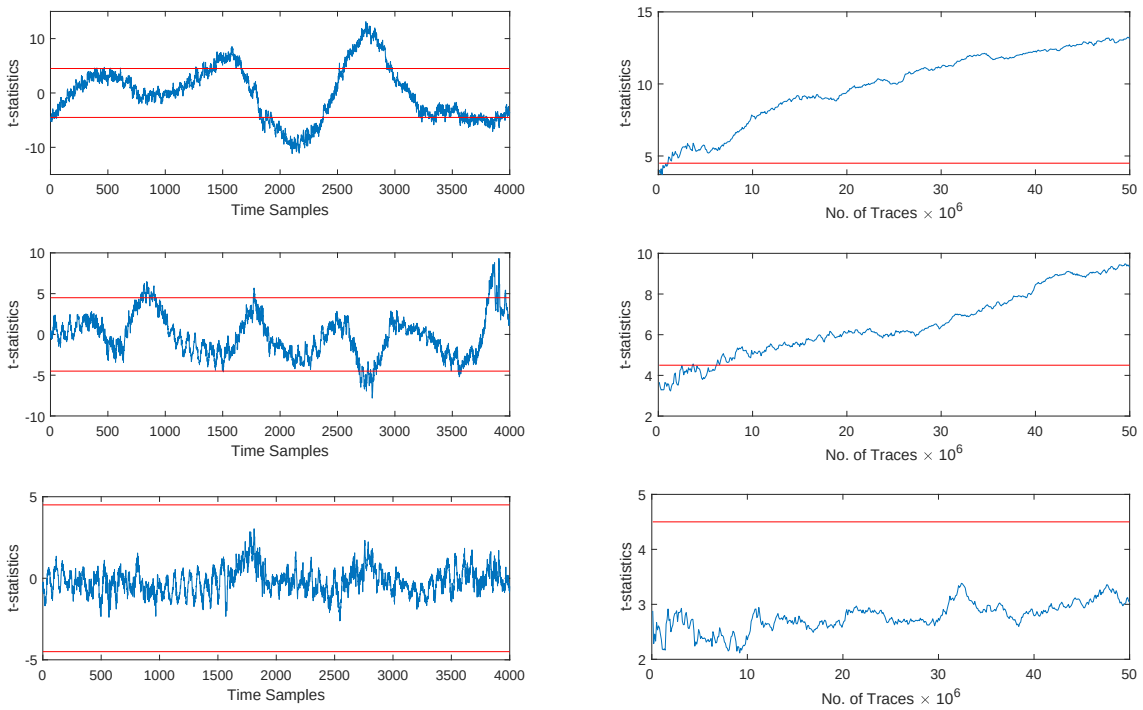


Figure 4.13: 90 nm ASIC, PRNG on, clock frequency 85 MHz (trojan triggered), t-test results with 50 million traces (left), absolute maximum t-value over the number of traces (right).

4.5.2.2 Results on 65 nm ASIC

After we have seen that the Trojan indeed achieves what it has been designed for on the 90 nm ASIC, we repeat the same kind of experiments on the 65 nm chip. At first, the results after 1,000,000 traces with the deactivated Trojan (25 MHz) and the switched off PRNG can be seen in Figure 4.15.

As before, detectable leakage is visible in all three statistical moments, but its magnitude is significantly smaller than on the 90 nm ASIC, indicating a lower signal-to-noise ratio. Thus, for the next step with PRNG on we measured more traces than before, namely 80,000,000. The results in Figure 4.16 show that with PRNG on and the Trojan not triggered at 25 MHz clock, there is no detectable leakage in any moment.

When measuring at 50 MHz, however, i.e., triggering the Trojan, significant leakage can be detected in all moments, as apparent in Figure 4.17.

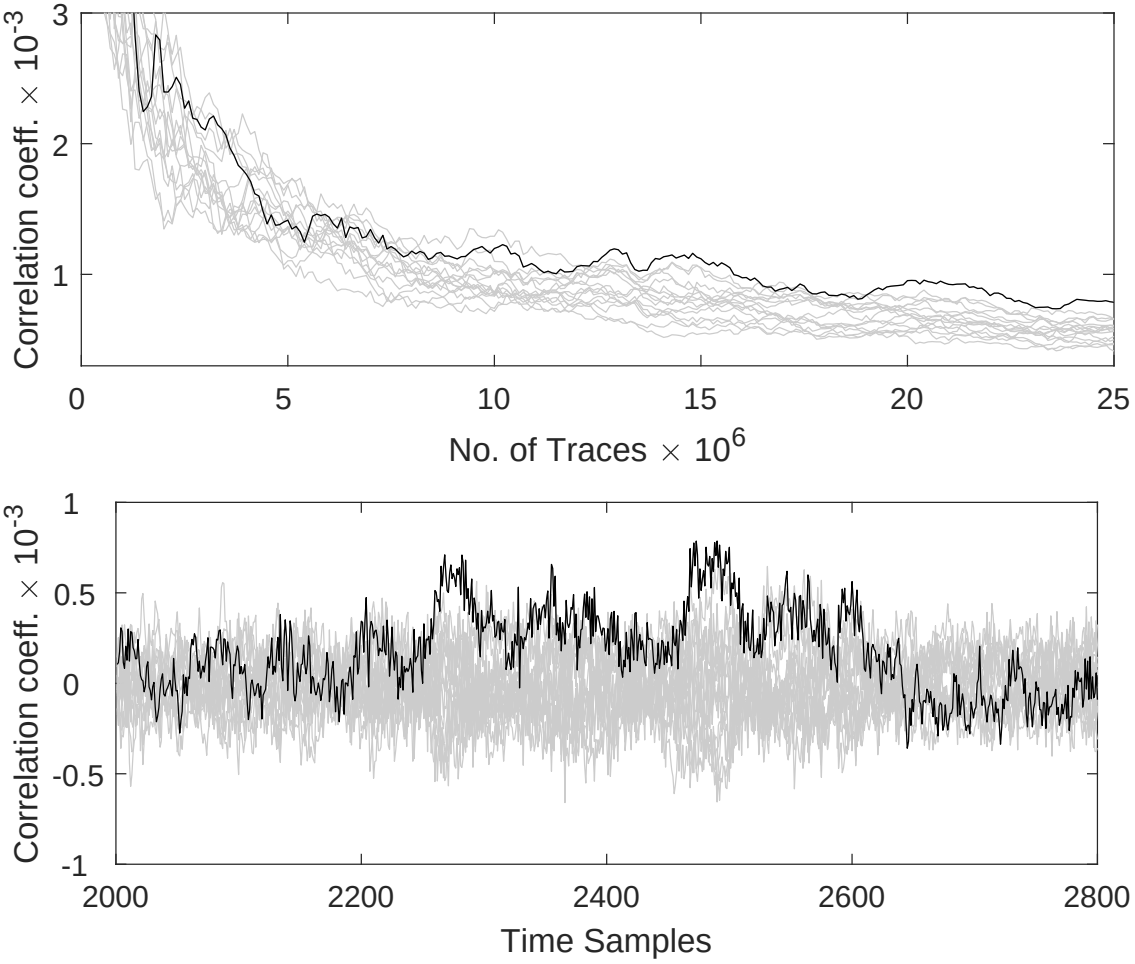


Figure 4.14: 90 nm ASIC, PRNG on, clock frequency 85 MHz (trojan triggered), CPA results targeting a key nibble based on an S-Box output bit with 25 million traces (right), absolute maximum correlation coefficient over the number of traces (left).

The successful CPA in 4.18 targeting a key nibble based on an S-Box output bit using 40,000,000 traces confirms that the leakage is indeed exploitable.

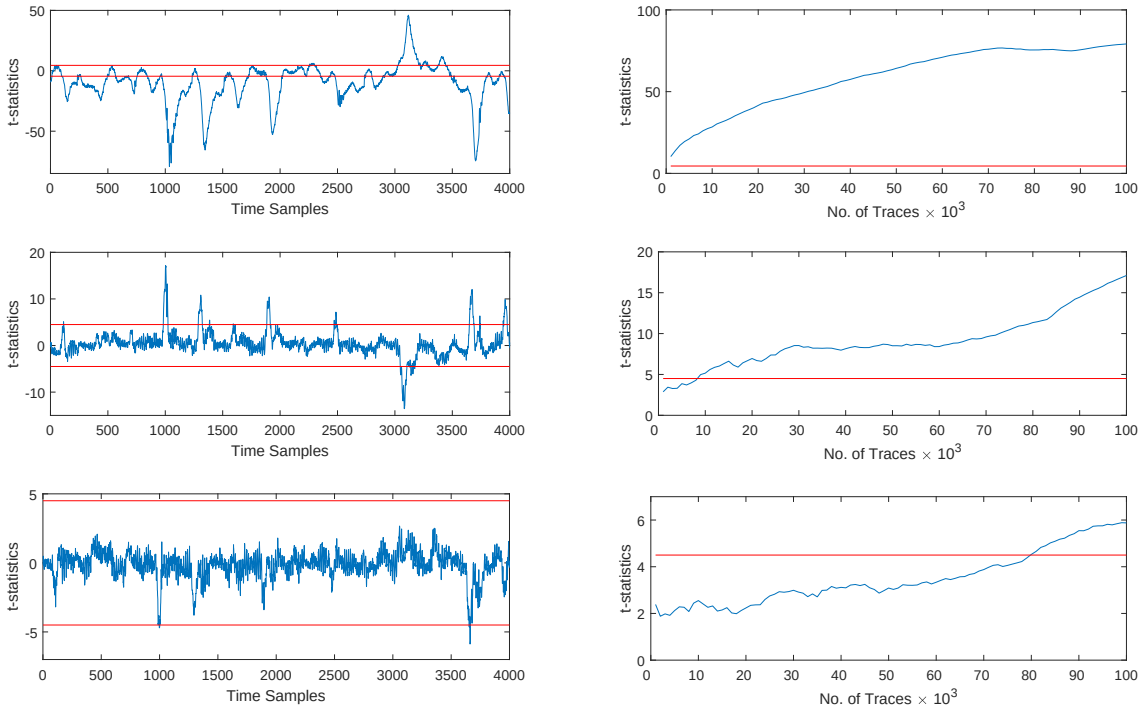


Figure 4.15: 65 nm ASIC, PRNG off, clock frequency 25 MHz (Trojan not triggered), t-test results with 1 million traces (left), absolute maximum t-value over the number of traces (right).

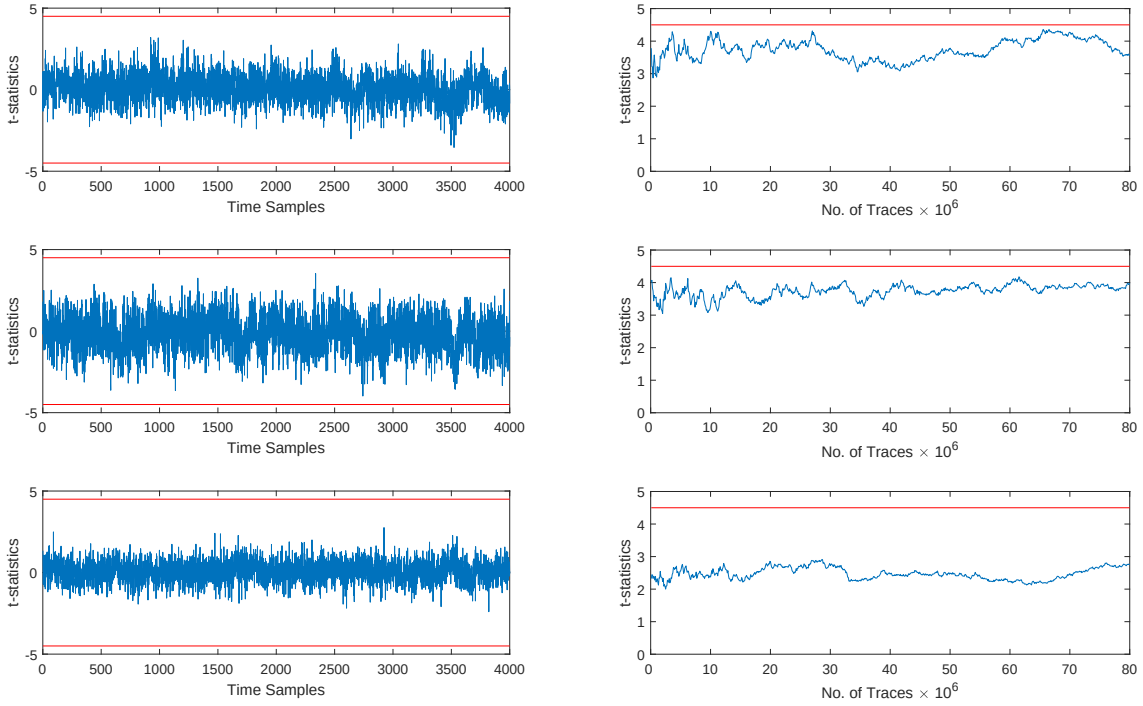


Figure 4.16: 65 nm ASIC, PRNG on, clock frequency 25 MHz (Trojan not triggered), t-test results with 80 million traces (left), absolute maximum t-value over the number of traces (right).

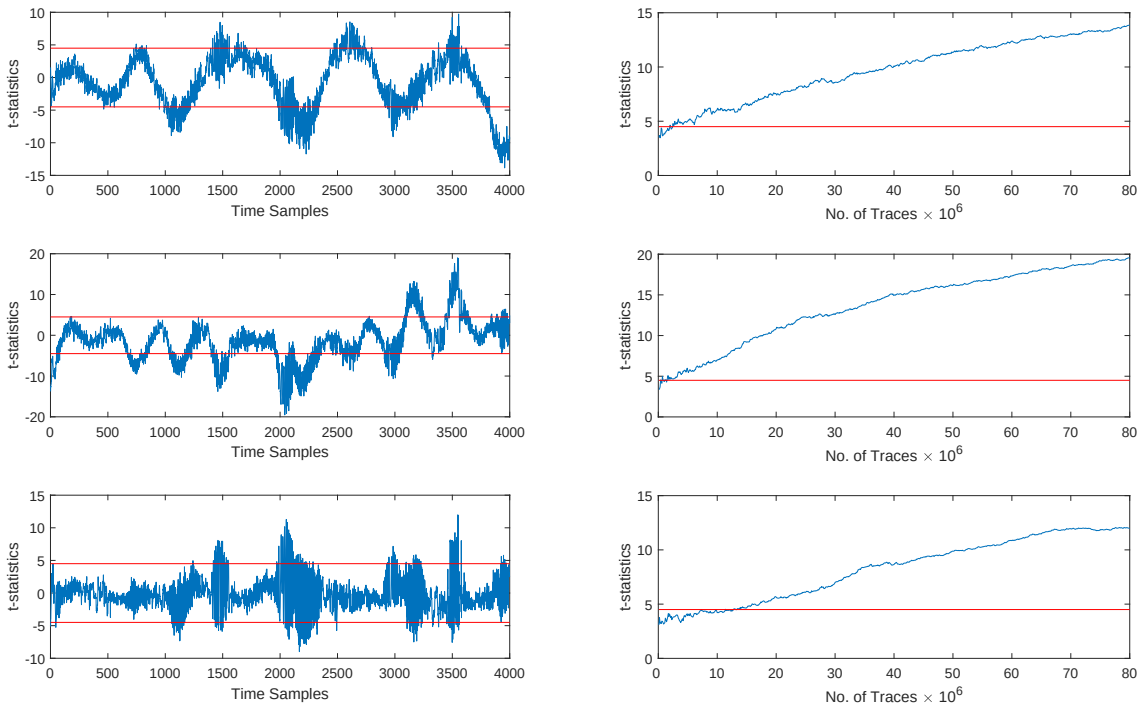


Figure 4.17: 65 nm ASIC, PRNG on, clock frequency 50 MHz (Trojan triggered), t-test results with 80 million traces (left), absolute maximum t-value over the number of traces (right).

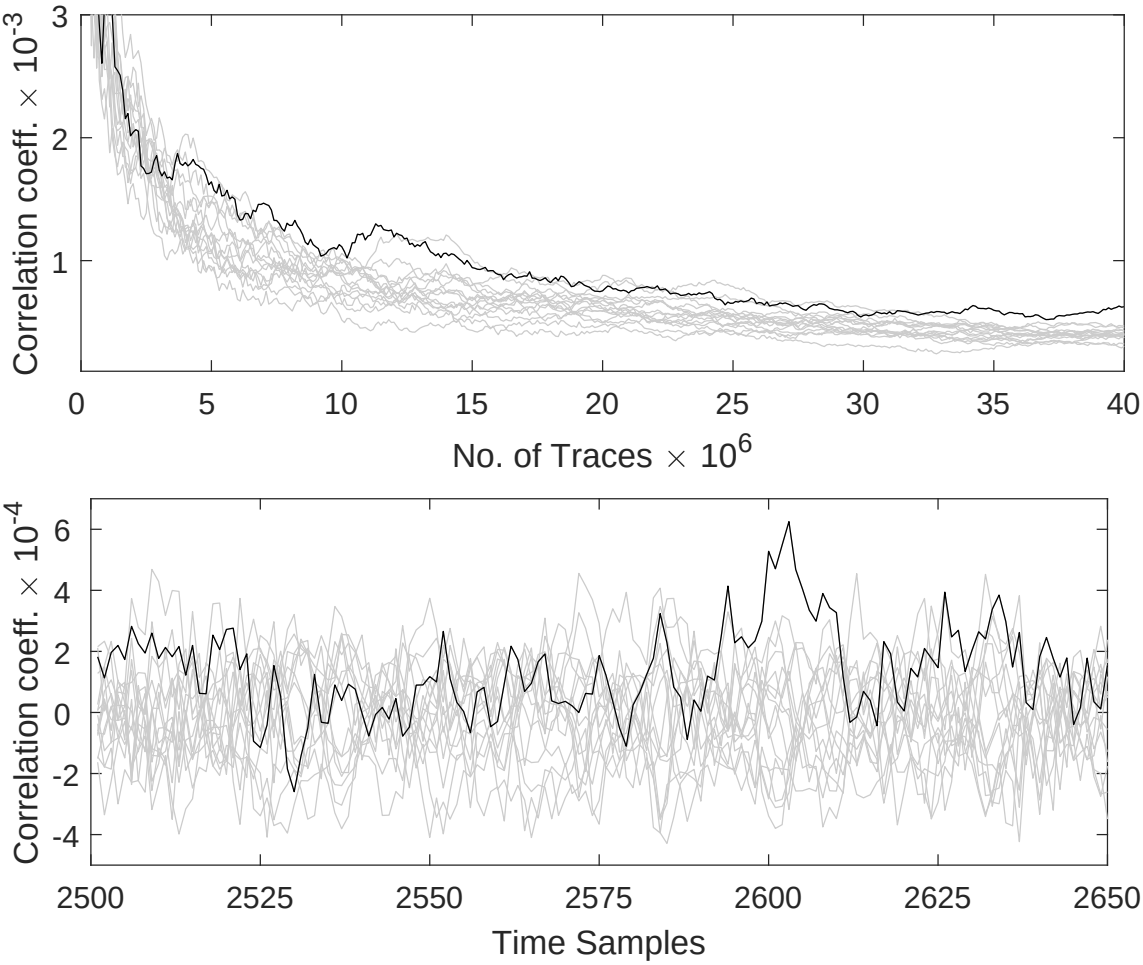


Figure 4.18: 65 nm ASIC, PRNG on, clock frequency 50 MHz (trojan triggered), CPA results targeting a key nibble based on an S-Box output bit with 40 million traces (right), absolute maximum correlation coefficient over the number of traces (left).

CHAPTER 5

TEMPERATURE-BASED HARDWARE TROJAN FOR RING-OSCILLATOR-BASED TRNGS

True random number generators (TRNGs) are essential components of cryptographic designs, which are used to generate private keys for encryption and authentication, and are used in masking countermeasures. In this work, we present a mechanism to design a stealthy parametric hardware Trojan for a specific TRNG architecture proposed by Yang et al. at ISSCC 2014. Once the Trojan is triggered the malicious TRNG generates predictable non-random outputs. Such a Trojan does not require any additional logic (even a single gate) and is purely based on subtle manipulations on the sub-transistor level. The underlying concept is to disable the entropy source at high temperature to trigger the Trojan, while ensuring that Trojan-infected TRNG works correctly under normal conditions. We show how an attack can be performed with the Trojan-infected TRNG design in which the attacker uses a stochastic Markov Chain model to predict its reduced-entropy outputs.

5.1 Introduction

High entropy random numbers are essential components for many cryptographic algorithms. Some applications of TRNGs are generating private keys, nonces, random numbers in challenge response protocols, and random numbers in countermeasure implementations to mask key-dependent values. One of the most popular method for generating random numbers is sampling jittery signals generated by ring oscillators (ROs) [86, 25]. In this chapter, we present a parametric hardware Trojan for an RO-based TRNG presented in [86] so that it works correctly under normal environmental

conditions, but produces predictable outputs at particular high temperatures. The Trojan is introduced by slightly changing the characteristics of a few transistors. We show that by injecting this Trojan, we are able to precisely control the output of the TRNG. This biasing can significantly lower the security level of any cryptographic applications that rely on the TRNG. A stochastic Markov Chain model allows the attacker to use their knowledge of the Trojan to predict the output of the Trojan-infected TRNG.

5.2 Ring oscillator-based TRNG

We consider the true random number generator (TRNG) design proposed in [86]. Figure 5.1 shows the TRNG architecture, which is based on the collapse time of three racing edges in a ring oscillator (RO). The design has two ring oscillators (RO). The first one is a reference that operates as a standard single-edge ring oscillator. The second one, which is called 3-edge RO, has three edges injected by three input nodes that propagate through the ring together at the same time (Figure 5.2). These edges in the 3-edge RO have same period, but they are shifted 120° in phase. As a result of this the frequency of the output of the 3-edge RO is boosted 3× in comparison to the regular RO. There is an increasing variation of the pulse width between edges in the 3-edge RO because of thermal noise (jitter) that exists in the system. This variation in the pulse widths causes neighboring edges to eventually collapse in the 3-edge RO, after which there is only a single oscillation in the ring. The collapse event in turn causes the 3-edge RO to change to a typical 1x frequency mode as can be seen in Figure 5.3. The time to collapse is used as the entropy source for the TRNG.

Phase frequency detector (PFD) module in the TRNG architecture shown in Figure 5.1 is used to detect the edge collapse events by comparing the frequencies of the regular RO and the 3-edge RO. A 14-bit counter counts the number of cycles until edge collapse event. This counter increments on rising edges of the 3-edge RO.

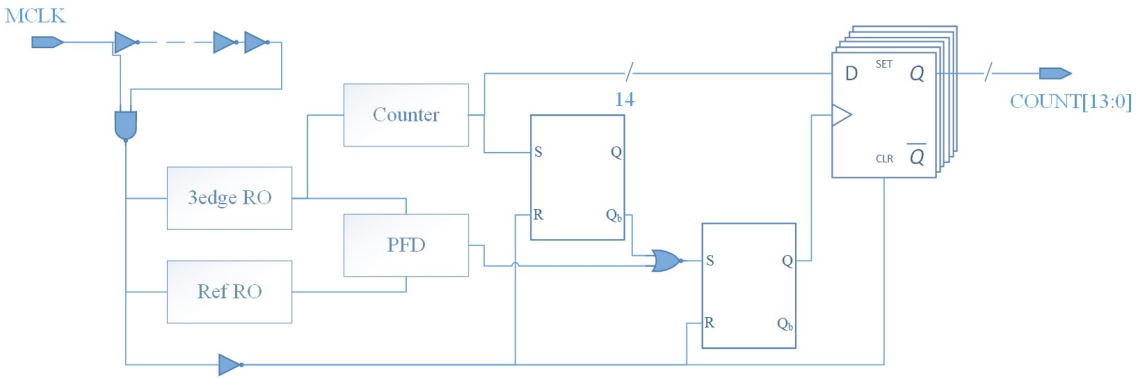


Figure 5.1: TRNG system block diagram [86]

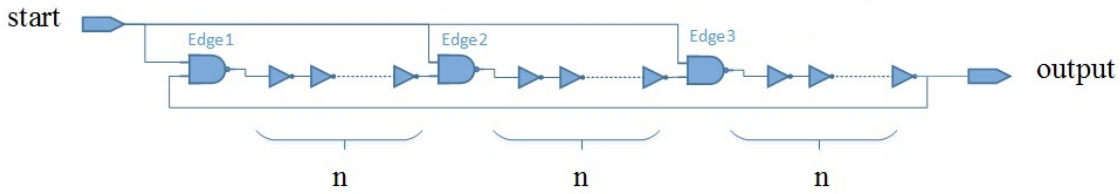


Figure 5.2: 3-edge ring oscillator

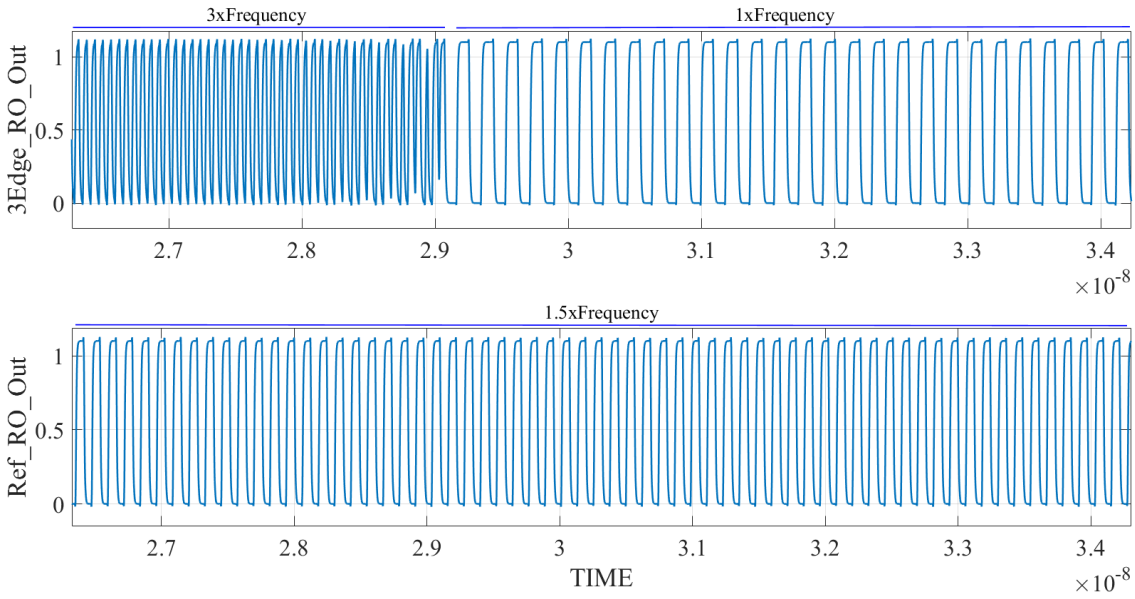


Figure 5.3: Output waveforms of the regular RO (bottom) and 3-edge RO (top)

The number of cycles to collapse follows inverse Gaussian distribution caused by thermal noise. In this design effect of process variation is canceled because all three edges propagate through the same RO stages[81]. We need to extract uniformly distributed random bits from collapse time. A simple method which has been applied in TRNG designs [81], [54] is to take the lower bits of the collapse count as output while the LSB is dropped to eliminate sensitivity to mismatch in the counter sampling flip-flop. In our work we consider COUNT[6:4] as the TRNG random output bits.

5.3 Hardware Trojan RO-based TRNG

Our goal is to maliciously manipulate the TRNG design to produce predictable outputs at a particular high environmental temperature. The conditions that cause a transition from correct behavior to Trojanized behavior should be available and known only to the Trojan attacker. In order to trigger the Trojan, the attacker must apply the specific temperature which could for example be beyond the maximum operating temperature of the device.

To realize such a scenario – inspired from the stealthy parametric Trojan introduced in [33] – we intentionally lengthen a certain path of a combinatorial circuit. This is done in such a way that by increasing the device's temperature, a signal on this path propagates slower than in normal operation. In the 3-edge RO construction, we achieve our goal of compromising the entropy by delaying one of the three edges of the 3-edge RO, which causes the RO to collapse in a few cycles with negligible variation. This rapid collapse behavior is not useful for generating random bits as it does not provide enough entropy.

Our technique for causing the delay change is based on manipulating one of the NAND gates that injects an edge to the RO circuit in such a way that its propagation delay is increased with temperature. The NAND gate must be very carefully altered in such a way that its propagation delay becomes more sensitive to the temperature

variation than the other gates of the 3-edge RO. Note that the functionality of the design is unaltered during the normal environmental temperature.

In the rest of this section, we explain how we inject the Trojan into the RO-based TRNG by modifying parameters of a few transistors, and how we use temperature as the trigger of our Trojan.

5.3.1 Temperature Dependence of Propagation Delay

Temperature can affect various process parameters of a device such as threshold voltage, carrier mobility, and leakage current. In this work, we focus on manipulation of threshold voltage and show how this can be used by an attacker to trigger the Trojan at a specific operating temperature. The threshold voltage of a device can be changed by various methods such as ion implantation or body biasing.

Threshold voltage and mobility decrease as the temperature increases. As supply voltage (V_{dd}) scaled in new technology generation, the value of $|V_{GS} - V_{TH}|$ decreases. The smaller $|V_{GS} - V_{TH}|$ makes saturation current more sensitive to change in V_{TH}, which decreases when temperature increases. The larger V_{TH} incurs less current that makes the device slower. On the other hand, transition delay is related to the carrier mobility, which decreases when temperature rises. Therefore, the device performance depends on the racing condition of electron mobility and V_{TH} when temperature rises. Equation 5.1 shows the variation of propagation delay D_p [83].

$$D_p \propto \frac{C_{out} V_{dd}}{I - d} \equiv \frac{C_{out} V_{dd}}{\mu(T)(V_{dd} - V_{TH}(T))} \tag{5.1}$$

As the carrier mobility (μ) decreases, the performance degrades, while the decrease in threshold voltage V_{TH} makes the device faster. Therefore, to make the propagation delay of our target NAND gate in the 3-edge RO circuit more sensitive to the temperature increases, we manipulate the threshold voltages of its transistors and use a combination of high V_{TH} and low V_{TH} transistors for its implementation.

5.3.2 Injecting temperature-triggered Trojan into RO-based TRNG

The time to collapse is used as the entropy source for random number generation, and delaying the start of any edges will cause the output to be not random. We focus on a single NAND gate B shown in Figure 5.4. We realize the Trojan functionality by increasing the delay sensitivity of the NAND gate B to temperature increases without modifying the logic circuit. The Trojan is triggered by increasing the delay of the NAND gate B by increasing temperature, so that the edge B is injected to the RO with delay. It means that the neighboring edges of the edge B can reach it sooner than in the unmodified circuit. As a result of this, we will have a small time to collapse and hence reduced entropy. Utilizing the temperature characteristics described in Section 5.3.1, an attacker adjusts the threshold of individual transistors in the circuit so that the circuit works correctly at the normal environmental temperatures, but acts as a Trojan beyond a particular temperature.

We show in Figure 5.4 how we modify the transistor-level implementation of the targeted NAND gate B to make it more sensitive to the temperature. We use high threshold voltages (high-V_{TH}) for the NMOS and PMOS transistors connected to the start input, whose threshold voltages are increased from their standard values. As a result of this, both modified transistors will be slow to propagate the transitions on the start input to the output of the NAND gate when the temperature increases. Furthermore, to these two transistors more sensitive to temperature than the other transistors, we use low threshold voltages (low-V_{TH}) for the rest of the transistors in the circuit so that their delay will not increased as much as these two targeted transistors. Note that the amount of delay added to the targeted NAND gate by the threshold voltage manipulation is small in the regular environmental temperature and does not affect the behavior of the 3-edge RO, so the malicious modification is extremely difficult to detect.

As an example, we simulate the maliciously manipulated 3-edge RO design in two different environmental temperatures; 25°C (as a normal environmental temperature),

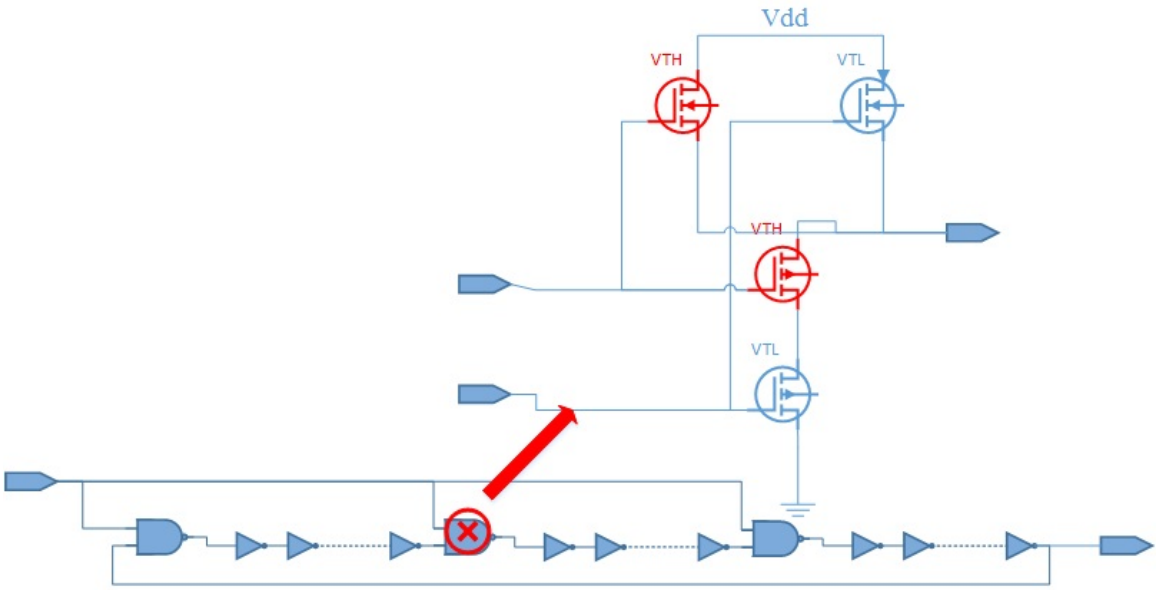


Figure 5.4: Threshold voltage manipulation of the 3-edge RO

and 120°C (as an increased environmental temperature). The Trojanized circuit behaves similar at 25°C to the unmodified 3-edge RO and there is a large collapse time (Figure 5.5(a)) which can be used as a source of entropy for random number generation. At 120°C, the behavior of the Trojanized circuit is changed and it collapses in a few cycles (Figure 5.5(b)). The immediate collapse occurs because the edge at NAND gate B in the manipulated 3-edge RO is not injected into the ring simultaneously with the two other edges injected at A and C. The immediate collapse behavior is not useful for extracting random bits and does not provide enough entropy. This is how the proposed temperature-triggered hardware Trojan removes the source of randomness from the 3-edge RO when the temperature rises.

5.4 How to Predict the Output of the Trojan TRNG

In this section, we describe how, in principle, an attack on the Trojan infected random number generator can be executed. When an attacker wants to attack the TRNG, she may choose the environment temperature and the input master clock (MCLK) of the TRNG at her will. But even when attacker knows the operating conditions of

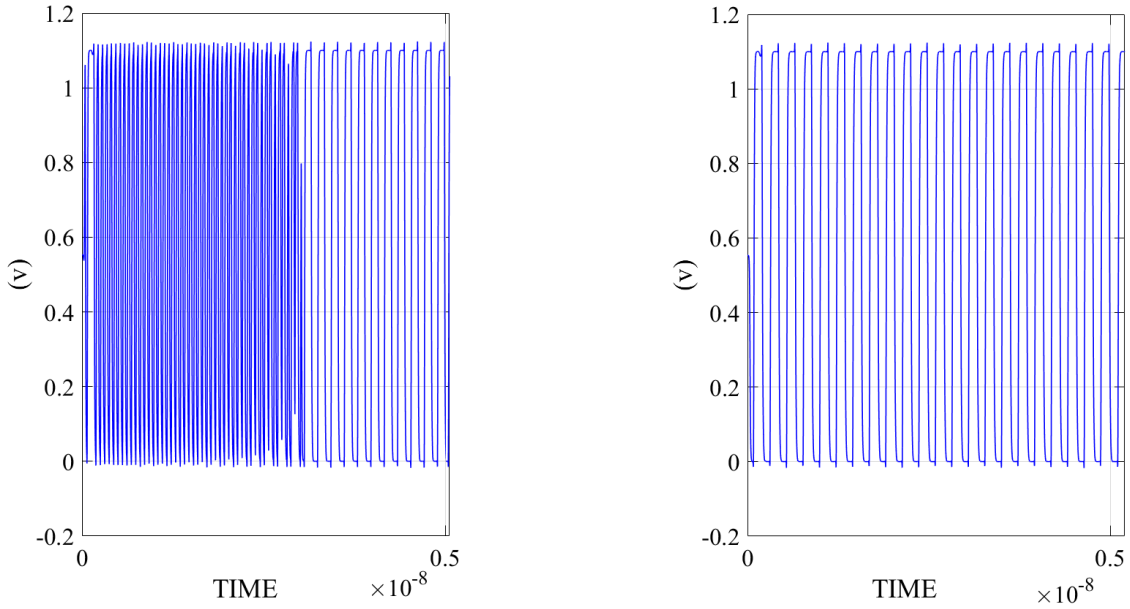


Figure 5.5: Output waveform of the Trojan infected 3-edge RO at (left) 25°C where there is a large time to collapse, and (right) 120°C where there is a very small time to collapse

the TRNG, its output bit-stream cannot be predicted perfectly, because of existing jitter in the TRNG, which follows independent normal distribution ($N(0, \sigma^2_{jitter})$). We elaborate a stochastic model for the attacker's knowledge to predict the output of the Trojan infected TRNG with a Markov chain model to describe the probability of occurrence for different output sequences of the Trojan infected TRNG.

5.4.1 Markov chain

A Markov chain is a stochastic model which describes a sequence of possible events in which the probability of each event depends only on the state in the previous event [28]. Assume we have a process with a set of states S = $s_1, s_2, ..., s_r$. The process starts in one of these states (initial state) and moves from one state to another. If the process is in state s_i, then it moves to state s_j with a transition probability p_{ij} at the next step , which is independent of states the chain was in before. Here we use two examples to explain the Markov chain concept from [35].

Example 1 [35]: Assume we want to model with a Markov chain the weather of a city that never has two nice days consecutively. If it has a nice day, it has snow or rain in the next day with equal probability. If it has snow or rain, it has an even chance of having the same the next day. If there is change from snow or rain, only half of the time is this a change to a nice day. The city transition probability matrix is defined as follows.

$$P = \begin{pmatrix} \frac{1}{2} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{2} \end{pmatrix}$$

Power Matrix: Let P be the transition matrix of a Markov chain. The ijth entry $p_{ij}^{(n)}$ of the matrix P^n gives the probability that the Markov chain, starting in state s_i, will be in state s_j after n steps [35].

Example 2 [35]: Consider the weather of the city explained in Example 1. We are interested in the state of the chain after a large number of steps. Here are the powers of the Transition Matrix P:

$$P^1 = \begin{pmatrix} 0.5 & 0.25 & 0.25 \\ 0.5 & 0.0 & 0.5 \\ 0.25 & 0.25 & 0.5 \end{pmatrix}$$

$$P^2 = \begin{pmatrix} 0.438 & 0.188 & 0.375 \\ 0.375 & 0.250 & 0.375 \\ 0.375 & 0.188 & 0.438 \end{pmatrix}$$

$$P^3 = \begin{pmatrix} 0.406 & 0.203 & 0.391 \\ 0.406 & 0.188 & 0.406 \\ 0.391 & 0.203 & 0.406 \end{pmatrix}$$

5.4.2 Predicting Bits

As we explained in Section 5.3, our Trojan removes the entropy source of the manipulated TRNG when temperature increases so that the Trojan infected TRNG counts as a non-random and predictable counter when temperature rises. For example the TRNG counter value increments by approximately 130 in each cycle when we set the period of the MCLK to 26ns. The variation in the counts is due to jitter which follows a normal distribution as shown in Figure 5.6 in which σ=100ps and x-axis shows the value that Trojan infected TRNG counts each time which makes the prediction hard for the attacker. Note that without loss of generality, we assumed σ equals to RO period for simplification.

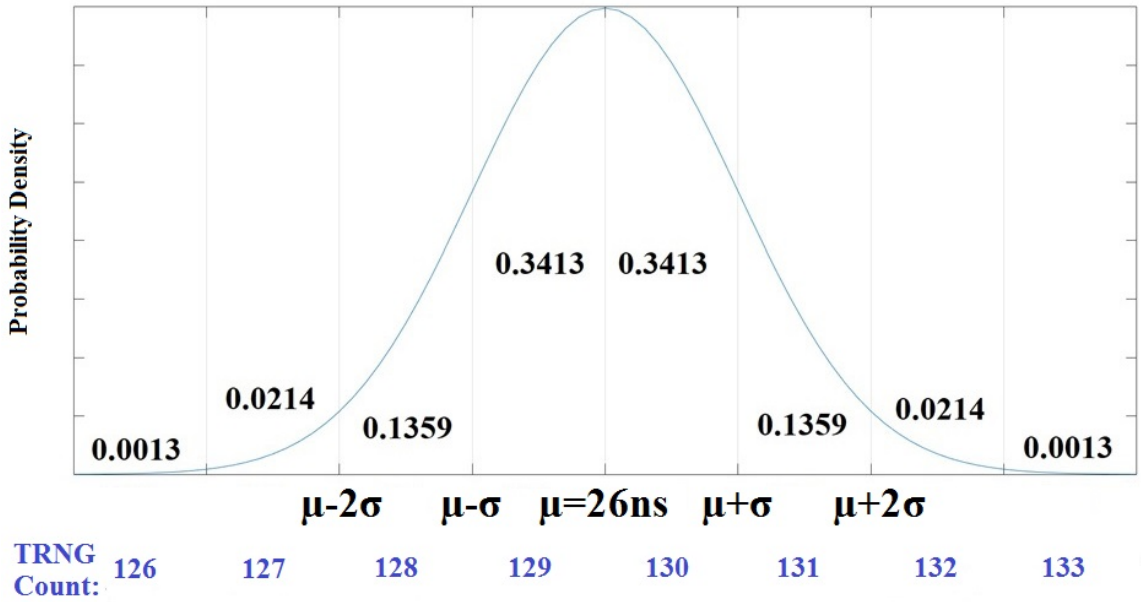


Figure 5.6: Jitter effect on the Trojan infected TRNG counter values

Transition matrix of the Trojan infected TRNG for seven lower output bits is shown by Equation 5.2 where p_{ij} is the transition probability that a TRNG output value which is currently i will move to value j at the next step. For example, $p_{01} = 0.341$ is the probability of TRNG output transition from 0000000 to 0000001. If the current output value of the TRNG is 0000000, in order to have the value 0000001 as the next output, the TRNG must increment its current value by 129 in the next clock cycle, which happens with probability of 0.341 based on Figure 5.6. As another example, consider $p_{10} = 0.021$ which is the probability of TRNG output transition from 0000001, to 0000000. If the current output value of the TRNG is 0000001, in order to have value 0000000 as the next value of the TRNG, the TRNG must increment its count by 127, which happens with probability 0.021 as shown in Figure 5.6.

$$P = \begin{array}{c} \\ 0000000 \\ 0000001 \\ \vdots \\ 1111111 \end{array} \begin{array}{c} \begin{array}{ccccc} 0000000 & 0000001 & \dots & 1111110 & 1111111 \end{array} \\ \begin{pmatrix} 0.136 & 0.341 & \dots & 0.001 & 0.021 \\ 0.021 & 0.136 & \dots & 0.000 & 0.001 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0.341 & 0.341 & \dots & 0.021 & 0.136 \end{pmatrix}_{128*128} \end{array} \tag{5.2}$$

The powers of the transition matrix of the Trojan infected TRNG give the attacker interesting information about the process as it evolves. She shall be particularly interested in the state of the chain after a large number of steps. For example, consider a scenario in which the TRNG output is used to produce a 15-bit secret key for a crypto system. Guessing this 15-bit secret key with certainty through brute force requires trying 2^{15} possible values for the key. An attacker that knows the properties of the output pattern of the Trojan infected TRNG, which are represented by the transition matrix and power matrices of the Trojan infected TRNG, can have an enhanced ability to predict output sequences.

The attacker, for guessing the 15-bit key generated by the Trojan infected TRNG, needs to predict 5 consecutive times the TRNG output ($COUNT[6:4]$). The power matrix P^4 gives the attacker the transition probabilities 5 steps from the current state of the TRNG output. However, the internal states between current state ($P^1 = P$) and the fifth state (P^4) of the TRNG output are also important for the attacker. Assume the attacker wants to find the probability with which TRNG generates sequence $000, 000, 000, 000, 000$. P^4 gives the probabilities with which TRNG generates output value = 000 at step 5 when its output value is 000 at step 1, independent of the output values in steps 2, 3, and 4. The attacker wants to know the probability that the intermediate output values (steps 2, 3, and 4) are 000 too. To solve this problem, we modify the transition matrix P before computing P^4 in order to avoid counting sequences that contain unwanted intermediate states. Equation 5.3 shows the modified P for sequence $000, 000, 000, 000, 000$ in which we set to zero the probabilities of all unwanted transitions that are incompatible with the desired sequence. For example, transition from state 0000001 to state 1111111 corresponds to $COUNT[6:4] = 000$ being followed by $COUNT[6:4] = 111$ which is incompatible with the target sequence, so we set the transition probability to zero so that it won't be counted. As can be seen in this figure, only a block of size 16×16 remains as non-zero; this 16×16 block denotes the probabilities of all possible transitions from states 000xxxx to states 000xxxx where $x \in \{0, 1\}$. After obtaining the modified transition matrix P', we compute P'^4 which includes the probabilities of four transitions from the current state.

$$
P' = \begin{array}{c} \\ 0000000 \\ 0000001 \\ \vdots \\ 0001111 \\ 0010000 \\ \vdots \\ 1111111 \end{array}
\begin{array}{c}
\begin{array}{ccccccc} 0000000 & 0000001 & \dots & 0001111 & 0010000 & \dots & 1111111 \end{array} \\
\left(\begin{array}{ccccccc}
0.136 & 0.341 & \dots & 0.000 & 0.000 & \dots & 0.000 \\
0.021 & 0.136 & \dots & 0.000 & 0.000 & \dots & 0.000 \\
\vdots & \vdots & \ddots & \vdots & 0.000 & \ddots & 0.000 \\
0.000 & 0.000 & \dots & 0.136 & 0.000 & \dots & 0.000 \\
0.000 & 0.000 & 0.000 & 0.000 & 0.000 & \dots & 0.000 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0.000 & 0.000 & \dots & 0.000 & 0.000 & \dots & 0.000
\end{array} \right)
\end{array}
\tag{5.3}
$$

Consider u as the probability vector which represents the initial state of a Markov chain, then the ith component of u represents the probability that the chain starts in state s_i. For our Trojan infected TRNG we assume all initial states are equally likely to occur. The following vector represents the initial state of our manipulated TRNG in which the probability that the chain starts in any state is $\frac{1}{128}$.

$$
u = [\frac{1}{128}\frac{1}{128}\frac{1}{128}...\frac{1}{128}\frac{1}{128}]_{1\times 128} \tag{5.4}
$$

The probability that the chain is in state s_i after n steps is the ith entry in the following vector:

$$
u^{(n)} = uP^n \tag{5.5}
$$

To obtain the probability of the sequence $000, 000, 000, 000, 000$ we set n = 4 in the Equation 5.5 and then add all non-zero probabilities as shown in Equations 5.6 and 5.7. The obtained value is almost equal to the measured value in our experiment.

$$u^{(4)} = uP'^4 = [\frac{1}{128}\frac{1}{128}\frac{1}{128}...\frac{1}{128}\frac{1}{128}]P'^4 \tag{5.6}$$

$$P(000,000,000,000,000) = \sum_{i=0}^{i=128} uP'^4[i] = 0.0764 \tag{5.7}$$

An attacker can use this method to obtain the most likely patterns for an n-bit key. Table 5.1 lists the eight most likely patterns of a 15-bit key and their probabilities. The attacker can guess the 15-bit key with the probability of 0.61 by trying these eight patterns.

Table 5.1: Most likely 15-bit patterns

15-bit Pattern	Probability
000000000000000	0.0764
001001001001001	0.0764
010010010010010	0.0764
011011011011011	0.0764
100100100100100	0.0764
101101101101101	0.0764
110110110110110	0.0764
111111111111111	0.0764

5.5 Practical Results

45nm Nangate Open Cell Library is used for our implementation of the Trojan free and Trojan infected TRNGs.

5.5.1 Randomness and Performance of the TRNG

The randomness of the Trojan free TRNG and the Trojan infected TRNG are evaluated by the NIST statistical test suite [5]. The Trojan free TRNG is robust and passes all NIST tests across all temperatures (25°C, 60°C, 120°C) as shown in Table 5.2. The NIST test suite results of the Trojan infected TRNG are also shown in Table 5.2 for different temperatures (25°C, 60°C, 120°C). The Trojan infected TRNG

passes the tests at the normal environmental temperatures (25°C, 60°C), but at the trigger temperature of 120°C does not pass the tests.

Table 5.2: NIST test suite results for Trojan free and Trojan infected TRNG

NIST	Trojan free design			Trojan infected design		
	25°C	60°C	120°C	25°C	60°C	120°C
Frequency	pass	pass	pass	pass	pass	pass
Block frequency	pass	pass	pass	pass	pass	fail
Cumulative sums (1)	pass	pass	pass	pass	pass	fail
Cumulative sums (2)	pass	pass	pass	pass	pass	pass
Longest runs	pass	pass	pass	pass	pass	pass
FFT	pass	pass	pass	pass	pass	fail
Approximate entropy	pass	pass	pass	pass	pass	fail

The measured distribution of number of cycles to collapse of the Trojan infected 3-edge RO at different environmental temperatures are shown in Figure 5.7 which follows inverse Gaussian distribution. Increasing the temperature causes the mean and variance of the number of cycles to collapse to decrease. At 120°C the mean value becomes 0 with negligible variance, meaning that the Trojan infected TRNG collapses within the first few cycles and therefore does not provide enough entropy.

Figure 5.8 illustrates the Trojan free TRNG bitstream and also the Trojan infected TRNG bitstream, raster scanning top-to-bottom then left-to-right. The outputs of the Trojan-free TRNG do not have any apparent pattern (Figure 5.8(a)), while the outputs of the Trojan infected TRNG at the trigger temperature are clearly periodic and non-random (Figure 5.8(b)). As another view of the same data, the output values of the Trojan-free TRNG for 600 samples (1800 bits) are shown in Figure 5.9(a), and the output values of the Trojan infected TRNG are shown in Figure 5.9(b). The Trojan infected TRNG produces output patterns that are largely periodic but have some noise.

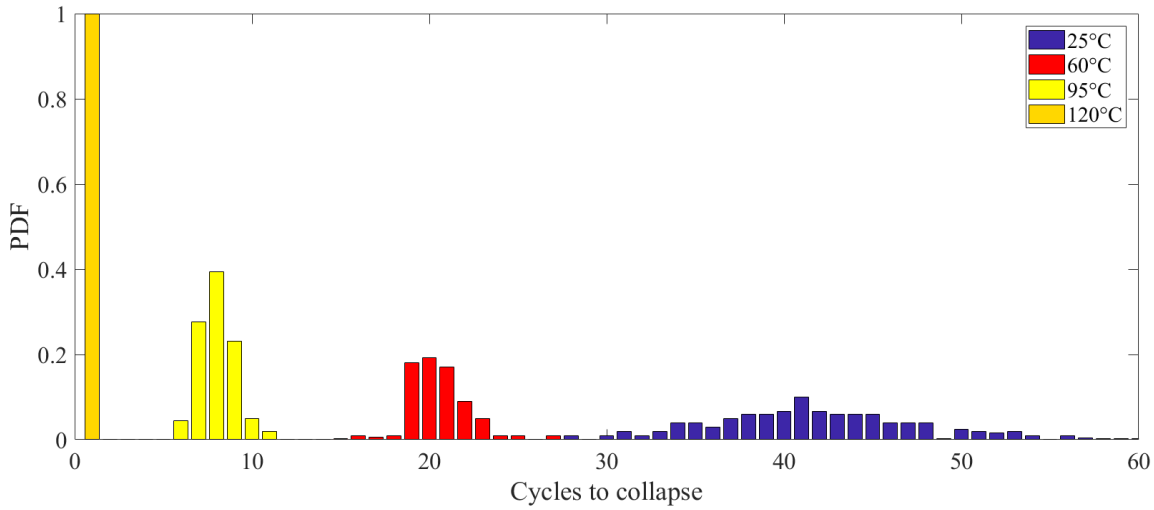
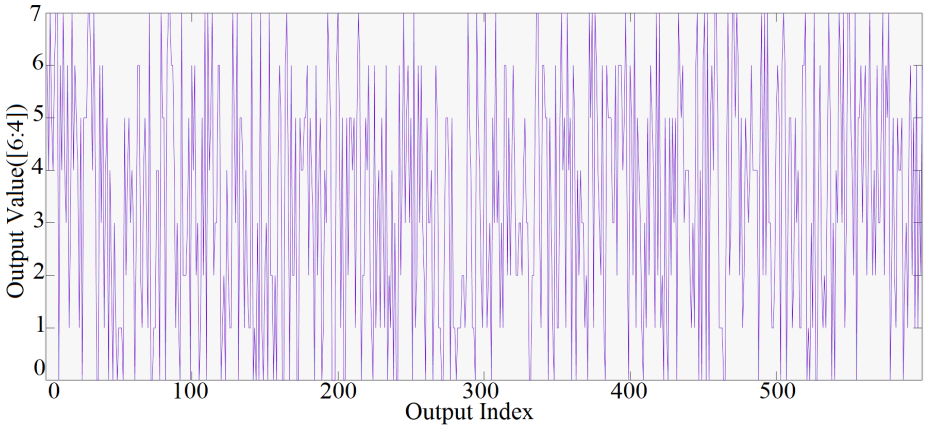


Figure 5.7: Distribution of 3-edge RO cycles to collapse at different environmental temperatures

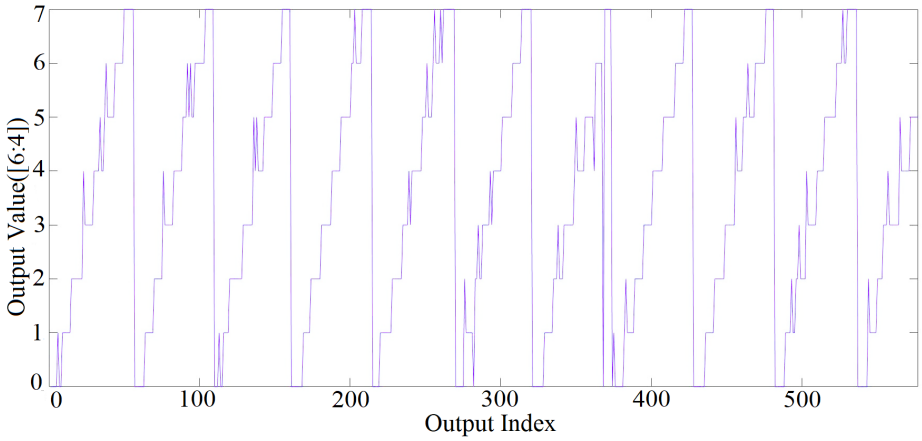
(a) Trojan free TRNG

(b) Trojan infected TRNG

Figure 5.8: Output patterns of (a) the Trojan free TRNG, and (b) the Trojan infected TRNG, raster scanning left-to-right then top-to-bottom.



(a)



(b)

Figure 5.9: Output values of (a) Trojan free TRNG and (b) Trojan infected TRNG at 120°C.

CHAPTER 6

CONCLUSION

Hardware Trojans have gained increasing attention in academia, industry and by government agencies. Designing reliable Trojan countermeasures requires an understanding of how hardware Trojans can be built in practice. This area, which has received relatively scant treatment in the literature, is the topic of this thesis. In particular, the thesis examines how particularly stealthy parametric Trojans can be introduced to VLSI circuits. Parametric Trojans are those which do not require any additional logic, but instead are based on subtle manipulations of designs at the sub-transistor level. The thesis has shown how parametric Trojans can infect three specific designs, for different purposes. All three Trojans proposed in the thesis would be very hard to detect, and may even be able to evade detection by a certification lab. The three specific Trojan examples in thesis are intended to be case studies, and the methodologies developed for inserting the Trojans can have broad application in other circuits.

Firstly, this thesis introduced a new type of parametric hardware Trojans based on rarely-sensitized path delay faults. While hardware Trojans using parametric changes (i.e. that only modify the performance/parameters of gates) have been proposed before, the previously proposed parametric hardware Trojans cannot be triggered deterministically. They are instead either triggered after time by aging [79], triggered randomly under reduced voltage [50] or are always on and can leak keys using a power side-channel [7]. In contrast, the proposed parametric hardware Trojan in this paper can be triggered by applying specific input sequences to the circuit. Hence, this work introduces the first trigger-based parametric hardware Trojan. To achieve

this, a SAT-based algorithm is presented which efficiently searches a combinational circuit for paths that are extremely rarely sensitized. A genetic algorithm is then used to distribute delays over all the gates on the path so that a path delay fault occurs when trigger inputs are applied, while for other inputs the timing criteria are met. In this way, a faulty response is computed only for a very small subset of input combinations. To demonstrate the usefulness of the proposed technique, a 32-bit multiplier is modified so that, for some multiplications, faulty responses are computed. These faults can be so rare that they do not interfere with normal operations but can still be used by the Trojan designer for a bug attack against public key algorithms. As a motivating example, we showed how this can be achieved for ECDH implementations. Please note that while we used a multiplier as our case study, the general idea of path delay Trojans and the tool-flow and algorithms presented in this work are not restricted to multipliers. Hence, this work shows that by making only extremely stealthy parametric changes to a design, a malicious factory could insert backdoors to leak out secret keys.

In this thesis, we also show how to insert a parametric hardware Trojan with very low overhead into SCA-resistant designs. The presented Trojan is capable of being integrated into both ASIC and FPGA platforms. Since it does not add any logic into the design, its chance of being detected is expected to be very low. Compared to the original design, its only footprint is around 10% decrease in the maximum clock frequency. We have shown that by increasing the clock frequency, the malicious threshold implementation design starts leaking exploitable information through side channels. Hence, the Trojan adversary can trigger the Trojan and make use of the exploitable leakage, while the design can pass SCA evaluations when the Trojan is not triggered. More precisely, suppose that the maximum clock frequency of the malicious device is 196 MHz. Hence, in an evaluation lab its SCA leakage will not be examined at 200 MHz because the device does not operate correctly. However, the Trojan adversary runs the device at 216 MHz and the SCA leakage becomes exploitable. To the best of

our knowledge, compared to the previous works in the areas of side-channel hardware Trojans, our construction is the only one which is applied on a provably-secure SCA countermeasure, and is parametric with very low overhead.

Finally, this dissertation also shows how a parametric hardware Trojan with very low overhead can be inserted into RO-based TRNG designs. The underlying concept is based on removing source of entropy of the TRNG when Trojan is triggered in high temperature, while the malicious TRNG works correctly and generate random outputs in normal conditions. To inject the Trojan, we lengthen the certain path of combinatorial logic in the RO such that increasing the temperature can diminish the entropy of the of the circuit upon which the TRNG is based. We elaborate a stochastic model based on Markov Chain for the attacker's knowledge to predict the output of the Trojan infected TRNG. This parametric Trojan allows us to significantly lower the security level even of highly protected crypto-core implementations that are connected to the TRNG.

BIBLIOGRAPHY

[1] Genetic Algorithm. `http://www.mathworks.com/discovery/genetic-algorithm.html`. [Accessed: 2016-02-01].

[2] Side-channel AttacK User Reference Architecture. `http://satoh.cs.uec.ac.jp/SAKURA/index.html`.

[3] Agrawal, D., Baktir, S, Karakoyunlu, D., Rohatgi, P., and Sunar, B. Trojan Detection using IC Fingerprinting. In *IEEE Symposium on Security and Privacy (SP 2007)* (2007), pp. 296–310.

[4] Bao, Chongxi, Forte, D., and Srivastava, A. On reverse engineering-based hardware trojan detection. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 35*, 1 (Jan 2016), 49–57.

[5] Bassham, Lawrence E, Rukhin, Andrew L, Soto, Juan, Nechvatal, James R, Smid, Miles E, Leigh, Stefan D, Levenson, M, Vangel, M, Heckert, Nathanael A, and Banks, D L. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Tech. rep., 2010.

[6] Bayon, Pierre, Bossuet, Lilian, Aubert, Alain, Fischer, Viktor, Poucheret, François, Robisson, Bruno, and Maurine, Philippe. Contactless electromagnetic active attack on ring oscillator based true random number generator. vol. 7275 of *Lecture Notes in Computer Science*, Springer, pp. 151–166.

[7] Becker, Georg T, Regazzoni, Francesco, Paar, Christof, and Burleson, Wayne P. Stealthy dopant-level hardware trojans. In *Cryptographic Hardware and Embedded Systems-CHES 2013*. Springer, 2013, pp. 197–214.

[8] Beyne, Tim, and Bilgin, Begül. Uniform First-Order Threshold Implementations. In *SAC 2016* (2017), Lecture Notes in Computer Science, Springer. to appear, `eprint.iacr.org/2016/715.pdf`.

[9] Bhunia, Swarup, Hsiao, Michael S, Banga, Mainak, and Narasimhan, Seetharam. Hardware trojan attacks: threat analysis and countermeasures. *Proceedings of the IEEE 102*, 8 (2014), 1229–1247.

[10] Biham, Eli, Carmeli, Yaniv, and Shamir, Adi. Bug attacks. In *Advances in Cryptology–CRYPTO 2008*. Springer, 2008, pp. 221–240.

[11] Biham, Eli, Carmeli, Yaniv, and Shamir, Adi. Bug attacks. *Journal of Cryptology* (2015), 1–31.

[12] Bilgin, Begül, Bogdanov, Andrey, Knezevic, Miroslav, Mendel, Florian, and Wang, Qingju. Fides: Lightweight Authenticated Cipher with Side-Channel Resistance for Constrained Hardware. In *CHES 2013* (2013), vol. 8086 of *Lecture Notes in Computer Science*, Springer, pp. 142–158.

[13] Bilgin, Begül, Daemen, Joan, Nikov, Ventzislav, Nikova, Svetla, Rijmen, Vincent, and Assche, Gilles Van. Efficient and First-Order DPA Resistant Implementations of Keccak. In *CARDIS 2013* (2014), vol. 8419 of *Lecture Notes in Computer Science*, Springer, pp. 187–199.

[14] Bilgin, Begül, Gierlichs, Benedikt, Nikova, Svetla, Nikov, Ventzislav, and Rijmen, Vincent. Higher-Order Threshold Implementations. In *ASIACRYPT 2014* (2014), vol. 8874 of *Lecture Notes in Computer Science*, Springer, pp. 326–343.

[15] Bilgin, Begül, Gierlichs, Benedikt, Nikova, Svetla, Nikov, Ventzislav, and Rijmen, Vincent. Trade-Offs for Threshold Implementations Illustrated on AES. *IEEE Trans. on CAD of Integrated Circuits and Systems 34*, 7 (2015), 1188–1200.

[16] Bilgin, Begül, Nikova, Svetla, Nikov, Ventzislav, Rijmen, Vincent, and Stütz, Georg. Threshold Implementations of All 3×3 and 4×4 S-Boxes. In *CHES 2012* (2012), vol. 7428 of *Lecture Notes in Computer Science*, Springer, pp. 76–91.

[17] Bilgin, Begül, Nikova, Svetla, Nikov, Ventzislav, Rijmen, Vincent, Tokareva, Natalia, and Vitkup, Valeriya. Threshold Implementations of Small S-boxes. *Cryptography and Communications 7*, 1 (2015), 3–33.

[18] Biryukov, Alex, Cannière, Christophe De, Braeken, An, and Preneel, Bart. A Toolbox for Cryptanalysis: Linear and Affine Equivalence Algorithms. In *EUROCRYPT 2003* (2003), vol. 2656 of *Lecture Notes in Computer Science*, Springer, pp. 33–50.

[19] Bogdanov, Andrey, Knudsen, Lars R., Leander, Gregor, Paar, Christof, Poschmann, Axel, Robshaw, Matthew J. B., Seurin, Yannick, and Vikkelsoe, C. PRESENT: An Ultra-Lightweight Block Cipher. In *CHES 2007* (2007), vol. 4727 of *Lecture Notes in Computer Science*, Springer, pp. 450–466.

[20] Bozilov, Dusan, Bilgin, Begül, and Sahin, Haci Ali. A Note on 5-bit Quadratic Permutations' Classification. *IACR Trans. Symmetric Cryptol. 2017*, 1 (2017), 398–404.

[21] Brumley, Billy B, Barbosa, Manuel, Page, Dan, and Vercauteren, Frederik. Practical realisation and elimination of an ecc-related software bug attack. In *Topics in Cryptology–CT-RSA 2012*. Springer, 2012, pp. 171–186.

[22] Canright, D., and Batina, Lejla. A Very Compact "Perfectly Masked" S-Box for AES. In *ACNS 2008* (2008), vol. 5037 of *Lecture Notes in Computer Science*, pp. 446–459.

[23] Carlet, Claude, Danger, Jean-Luc, Guilley, Sylvain, and Maghrebi, Houssem. Leakage Squeezing of Order Two. In *INDOCRYPT 2012* (2012), vol. 7668 of *Lecture Notes in Computer Science*, Springer, pp. 120–139.

[24] Chakraborty, Rajat Subhra, Wolff, Francis, Paul, Somnath, Papachristou, Christos, and Bhunia, Swarup. Mero: A statistical approach for hardware trojan detection. In *Cryptographic Hardware and Embedded Systems-CHES 2009*. Springer, 2009, pp. 396–410.

[25] Cherkaoui, Abdelkarim, Fischer, Viktor, Fesquet, Laurent, and Aubert, Alain. A very high speed true random number generator with entropy assessment. In *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings* (2013), pp. 179–196.

[26] Eggersglüß, Stephan, Wille, Robert, and Drechsler, Rolf. Improved SAT-based ATPG: More constraints, better compaction. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2013), pp. 85–90.

[27] Ender, Maik, Ghandali, Samaneh, Moradi, Amir, and Paar, Christof. The First Thorough Side-Channel Hardware Trojan. In *ASIACRYPT 2017* (2017), vol. 10624 of *Lecture Notes in Computer Science*, Springer, pp. 755–780.

[28] Gagniuc, Paul A. *Markov chains: from theory to implementation and experimentation*. John Wiley & Sons, 2017.

[29] Ghandali, S., Alizadeh, B., and Navabi, Z. Low Power Scheduling in High-level Synthesis using Dual-Vth Library. In *16th International Symposium on Quality Electronic Design (ISQED)* (2015), pp. 507–511.

[30] Ghandali, Samaneh, Alizadeh, Bijan, Fujita, Masahiro, and Navabi, Zainalabedin. Rtl datapath optimization using system-level transformations. In *Fifteenth International Symposium on Quality Electronic Design* (2014), IEEE, pp. 309–316.

[31] Ghandali, Samaneh, Alizadeh, Bijan, Navabi, Zainalabedin, and Fujita, Masahiro. Polynomial datapath synthesis and optimization based on vanishing polynomial over z 2 m and algebraic techniques. In *Tenth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMCODE2012)* (2012), IEEE, pp. 65–74.

[32] Ghandali, Samaneh, Becker, Georg T., Holcomb, Daniel, and Paar, Christof. A Design Methodology for Stealthy Parametric Trojans and Its Application to Bug Attacks. In *CHES 2016* (2016), vol. 9813 of *Lecture Notes in Computer Science*, Springer, pp. 625–647.

[33] Ghandali, Samaneh, Moos, Thorben, Moradi, Amir, and Paar, Christof. Side-channel hardware trojan for provably-secure sca-protected implementations. In *submitted to IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, IEEE.

[34] Goodwill, G., Jun, B., Jaffe, J., and Rohatgi, P. A testing methodology for side channel resistance validation. In *NIST non-invasive attack testing workshop* (2011). `http://csrc.nist.gov/news_events/non-invasive-attack-testing-workshop/papers/08_Goodwill.pdf`.

[35] Grinstead, Charles M, and Snell, James Laurie. *Introduction to probability*. American Mathematical Soc., 2012.

[36] Gross, Hannes, Mangard, Stefan, and Korak, Thomas. An Efficient Side-Channel Protected AES Implementation with Arbitrary Protection Order. In *CT-RSA 2017* (2017), vol. 10159 of *Lecture Notes in Computer Science*, Springer, pp. 95–112.

[37] Groß, Hannes, Wenger, Erich, Dobraunig, Christoph, and Ehrenhöfer, Christoph. Suit up! - Made-to-Measure Hardware Implementations of ASCON. In *DSD 2015* (2015), IEEE Computer Society, pp. 645–652.

[38] Güneysu, Tim, and Moradi, Amir. Generic Side-Channel Countermeasures for Reconfigurable Devices. In *CHES 2011* (2011), vol. 6917 of *Lecture Notes in Computer Science*, Springer, pp. 33–48.

[39] Gupta, Puneet, Kahng, Andrew B., Sharma, Puneet, and Sylvester, Dennis. Gate-length biasing for runtime-leakage control. *IEEE Trans. on CAD of Integrated Circuits and Systems 25*, 8 (2006), 1475–1485.

[40] Heragu, K., Agrawal, V.D., and Bushnell, M.L. FACTS: fault coverage estimation by test vector sampling. In *Proceedings of IEEE VLSI Test Symposium* (1994), pp. 266–271.

[41] Hicks, M., Finnicum, M., King, S. T., Martin, M. MK, and Smith, J. M. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *IEEE Symposium on Security and Privacy (SP 2010)* (2010), pp. 159–172.

[42] Karri, Ramesh, Rajendran, Jeyavijayan, Rosenfeld, Kurt, and Tehranipoor, Mohammad. Trustworthy hardware: Identifying and classifying hardware trojans. *Computer*, 10 (2010), 39–46.

[43] Kasper, Markus, Moradi, Amir, Becker, Georg T., Mischke, Oliver, Güneysu, Tim, Paar, Christof, and Burleson, Wayne. Side channels as building blocks. *J. Cryptographic Engineering 2*, 3 (2012), 143–159.

[44] Keshavarz, Shahrzad, and Holcomb, Daniel. Threshold-based obfuscated keys with quantifiable security against invasive readout. In *Proceedings of the 36th International Conference on Computer-Aided Design* (2017), IEEE Press, pp. 57–64.

[45] Keshavarz, Shahrzad, Paar, Christof, and Holcomb, Daniel. Design automation for obfuscated circuits with multiple viable functions. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017* (2017), IEEE, pp. 886–889.

[46] King, S. T., Tucek, J., Cozzie, A., Grier, C., Jiang, W., and Zhou, Y. Designing and implementing malicious hardware. In *Proceedings of the 1st USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET 08)* (2008), pp. 1–8.

[47] Kocher, Paul C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO 1996* (1996), vol. 1109 of *Lecture Notes in Computer Science*, Springer, pp. 104–113.

[48] Kocher, Paul C., Jaffe, Joshua, and Jun, Benjamin. Differential Power Analysis. In *CRYPTO 1999* (1999), vol. 1666 of *Lecture Notes in Computer Science*, Springer, pp. 388–397.

[49] Kulkarni, S. H., Sylvester, D. M., and Blaauw, D. T. Design-time optimization of post-silicon tuned circuits using adaptive body bias. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27*, 3 (March 2008), 481–494.

[50] Kumar, Raghavan, Jovanovic, Philipp, Burleson, Wayne, and Polian, Ilia. Parametric trojans for fault-injection attacks on cryptographic hardware. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)* (2014), IEEE, pp. 18–28.

[51] Lin, L., Kasper, M., Güneysu, T., Paar, C., and Burleson, W. Trojan Side-Channels: Lightweight Hardware Trojans through Side-Channel Engineering. In *Cryptographic Hardware and Embedded Systems - CHES 2009* (2009), LNCS, Springer, pp. 382–395.

[52] Lin, Lang, Burleson, Wayne, and Paar, Christof. MOLES: Malicious off-chip leakage enabled by side-channels. In *ICCAD 2009* (2009), ACM, pp. 117–122.

[53] Lin, Lang, Kasper, Markus, Güneysu, Tim, Paar, Christof, and Burleson, Wayne. Trojan Side-Channels: Lightweight Hardware Trojans through Side-Channel Engineering. In *CHES 2009* (2009), vol. 5747 of *Lecture Notes in Computer Science*, Springer, pp. 382–395.

[54] Liu, Nurrachman, Pinckney, Nathaniel, Hanson, Scott, Sylvester, Dennis, and Blaauw, David. A true random number generator using time-dependent dielectric breakdown. In *2011 Symposium on VLSI Circuits-Digest of Technical Papers* (2011), IEEE, pp. 216–217.

[55] Maghrebi, Houssem, Guilley, Sylvain, and Danger, Jean-Luc. Leakage Squeezing Countermeasure against High-Order Attacks. In *WISTP 2011* (2011), vol. 6633 of *Lecture Notes in Computer Science*, Springer, pp. 208–223.

[56] Mangard, Stefan, Oswald, Elisabeth, and Popp, Thomas. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.

[57] Mangard, Stefan, Pramstaller, Norbert, and Oswald, Elisabeth. Successfully Attacking Masked AES Hardware Implementations. In *CHES 2005* (2005), vol. 3659 of *Lecture Notes in Computer Science*, Springer, pp. 157–171.

[58] Markettos, A. Theodore, and Moore, Simon W. The frequency injection attack on ring-oscillator-based true random number generators. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings* (2009), pp. 317–331.

[59] Martin, Honorio, Korak, Thomas, San Millán, Enrique, and Hutter, Michael. Fault attacks on strngs: Impact of glitches, temperature, and underpowering on randomness. *IEEE transactions on information forensics and security 10*, 2 (2015), 266–277.

[60] Moradi, Amir, Kirschbaum, Mario, Eisenbarth, Thomas, and Paar, Christof. Masked Dual-Rail Precharge Logic Encounters State-of-the-Art Power Analysis Methods. *IEEE Trans. VLSI Syst. 20*, 9.

[61] Moradi, Amir, Mischke, Oliver, and Eisenbarth, Thomas. Correlation-Enhanced Power Analysis Collision Attack. In *CHES 2010* (2010), vol. 6225 of *Lecture Notes in Computer Science*, Springer, pp. 125–139.

[62] Moradi, Amir, Poschmann, Axel, Ling, San, Paar, Christof, and Wang, Huaxiong. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In *EUROCRYPT 2011* (2011), vol. 6632, Springer, pp. 69–88.

[63] Moradi, Amir, and Schneider, Tobias. Side-Channel Analysis Protection and Low-Latency in Action - - Case Study of PRINCE and Midori. In *ASIACRYPT 2016* (2016), vol. 10031 of *Lecture Notes in Computer Science*, pp. 517–547.

[64] Moradi, Amir, and Wild, Alexander. Assessment of Hiding the Higher-Order Leakages in Hardware - What Are the Achievements Versus Overheads? In *CHES 2015* (2015), vol. 9293 of *Lecture Notes in Computer Science*, Springer, pp. 453–474.

[65] Nikova, Svetla, Rijmen, Vincent, and Schläffer, Martin. Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches. *J. Cryptology 24*, 2 (2011), 292–321.

[66] Oswald, Elisabeth, Mangard, Stefan, Pramstaller, Norbert, and Rijmen, Vincent. A Side-Channel Analysis Resistant Description of the AES S-Box. In *FSE 2005* (2005), vol. 3557 of *Lecture Notes in Computer Science*, Springer, pp. 413–423.

[67] Popp, Thomas, Kirschbaum, Mario, Zefferer, Thomas, and Mangard, Stefan. Evaluation of the Masked Logic Style MDPL on a Prototype Chip. In *CHES 2007* (2007), vol. 4727 of *Lecture Notes in Computer Science*, Springer, pp. 81–94.

[68] Poschmann, Axel, Moradi, Amir, Khoo, Khoongming, Lim, Chu-Wee, Wang, Huaxiong, and Ling, San. Side-Channel Resistant Crypto for Less than 2, 300 GE. *J. Cryptology 24*, 2 (2011), 322–345.

[69] Pozo, Santos Merino Del, and Standaert, François-Xavier. Getting the Most Out of Leakage Detection - Statistical tools and Measurement Setups Hand in Hand. In *COSADE 2017* (2017), Lecture Notes in Computer Science, Springer. to appear.

[70] Prouff, Emmanuel, Rivain, Matthieu, and Bevan, Régis. Statistical Analysis of Second Order Differential Power Analysis. *IEEE Trans. Computers 58*, 6 (2009), 799–811.

[71] Rajendran, J., Jyothi, V., and Karri, R. Blue team red team approach to hardware trust assessment. In *IEEE 29th International Conference on Computer Design (ICCD 2011)* (oct. 2011), pp. 285–288.

[72] Rajendran, J., Jyothi, V., Sinanoglu, O., and Karri, R. Design and analysis of ring oscillator based Design-for-Trust technique. In *29th IEEE VLSI Test Symposium (VTS 2011)* (2011), pp. 105–110.

[73] Reparaz, Oscar, Bilgin, Begül, Nikova, Svetla, Gierlichs, Benedikt, and Verbauwhede, Ingrid. Consolidating Masking Schemes. In *CRYPTO 2015* (2015), vol. 9215 of *Lecture Notes in Computer Science*, Springer, pp. 764–783.

[74] Saha, Sayandeep, Chakraborty, Rajat S., Nuthakki, Srinivasa S., and Mukhopadhyay, Debdeep. Improved test pattern generation for hardware trojan detection using genetic algorithm and boolean satisfiability. In *Cryptographic Hardware and Embedded Systems–CHES 2015*. Springer, 2015, pp. 577–596.

[75] Sasdrich, Pascal, and Güneysu, Tim. Implementing curve25519 for side-channel–protected elliptic curve cryptography. *ACM Transactions on Reconfigurable Technology and Systems (TRETS) 9*, 1 (2015), 3.

[76] Sasdrich, Pascal, Moradi, Amir, and Güneysu, Tim. Affine Equivalence and Its Application to Tightening Threshold Implementations. In *SAC 2015* (2015), vol. 9566 of *Lecture Notes in Computer Science*, Springer, pp. 263–276.

[77] Sasdrich, Pascal, Moradi, Amir, and Güneysu, Tim. Hiding Higher-Order Side-Channel Leakage - Randomizing Cryptographic Implementations in Reconfigurable Hardware. In *CT-RSA 2017* (2017), vol. 10159 of *Lecture Notes in Computer Science*, Springer, pp. 131–146.

[78] Schneider, Tobias, and Moradi, Amir. Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations. In *CHES 2015* (2015), vol. 9293 of *Lecture Notes in Computer Science*, Springer, pp. 495–513.

[79] Shiyanovskii, Y., Wolff, F., Rajendran, A., Papachristou, C., Weyer, D., and Clay, W. Process reliability based trojans through NBTI and HCI effects. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2010)* (2010), pp. 215–222.

[80] Sugawara, Takeshi, Suzuki, Daisuke, Fujii, Ryoichi, Tawa, Shigeaki, Hori, Ryohei, Shiozaki, Mitsuru, and Fujino, Takeshi. Reversing stealthy dopant-level circuits. In *Cryptographic Hardware and Embedded Systems–CHES 2014*. Springer, 2014, pp. 112–126.

[81] Tang, Qianying, Kim, Bongjin, Lao, Yingjie, Parhi, Keshab K., and Kim, Chris H. True random number generator circuits based on single- and multi-phase beat frequency detection. In *Proceedings of the IEEE 2014 Custom Integrated Circuits Conference, CICC 2014, San Jose, CA, USA, September 15-17, 2014* (2014), pp. 1–4.

[82] Tang, X., Zhou, H., and Banerjee, P. Leakage Power Optimization With Dual-Vth Library In High-Level Synthesis. In *42nd annual Design Automation Conference (DAC 2005)* (2005), pp. 202–207.

[83] Verma, Preeti, and Mishra, RA. Temperature dependence of propagation delay characteristic in lector based cmos circuit. *IJCA Special Issue on Electronics, Information and Communication Engineering ICEICE* (2011), 28–30.

[84] Waksman, A., and Sethumadhavan, S. Silencing hardware backdoors. In *IEEE Symposium on Security and Privacy (SP 2011)* (2011), pp. 49–63.

[85] Wold, Knut, and Tan, Chik How. Analysis and enhancement of random number generator in FPGA based on oscillator rings. *Int. J. Reconfig. Comp. 2009* (2009), 501672:1–501672:8.

[86] Yang, Kaiyuan, Fick, David, Henry, Michael B, Lee, Yoonmyung, Blaauw, David, and Sylvester, Dennis. 16.3 a 23mb/s 23pj/b fully synthesized true-random-number generator in 28nm and 65nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International* (2014), IEEE, pp. 280–281.

[87] Yier, J., and Makris, Y. Hardware Trojan detection using path delay fingerprint. In *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST 2008)* (2008), pp. 51–57.