KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Design and Verification of Side-Channel and Fault Attacks Countermeasures

Victor Arribas

Supervisors:
Prof. dr. ir. Vincent Rijmen
Dr. Svetla Nikova

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

May 2020

Design and Verification of Side-Channel and Fault Attacks Countermeasures

Victor ARRIBAS

Examination committee:
Prof. dr. ir. Hugo Hens, chair
Prof. dr. ir. Vincent Rijmen, supervisor
Dr. Svetla Nikova, supervisor
Prof. dr. ir. Liesbet Van der Perre
Prof. dr. ir. Ingrid Verbauwhede
Prof. dr. Emmanuel Prouff
(Sorbonne Universités, ANSSI)
Prof. dr. François-Xavier Standaert
(Université Catholique de Louvain)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

May 2020

© 2020 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Victor Arribas, Kasteelpark Arenberg 10 - bus 2452, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

"Shikin Haramitsu Daikomyo"
— Ninjutsu motto

Acknowledgments

And there they go, four years of my life. It seems like yesterday that I arrived in Leuven to give my first seminar at COSIC. At that point, I was completely uncertain of my coming future, unaware of what was coming, unaware that the next four years would change my life not only professionally, but also at the personal level. Despite the unknown, the excitement and eagerness for learning were certain. That same day I got accepted to start my PhD in COSIC, and I still remember the pure happiness I felt with the news. Now, in hindsight, I can firmly say this was the right decision and no wonder why this time has gone so fast.

Dear Vincent, I would like to thank you for your guidance and support throughout these years. For your trust, for always been there when I knocked at the door of your office, and for bearing with my restlessness. As you once said, the crypto of the real world is not as exciting as that of Dan Brown, but I could certainly write a book (a second one!) with all the experiences I have had throughout these years, which helped me grow into who I am today.

Dear Svetla, you have always been there to listen to my problems or worries, whether it was personal or academic, to guide me with whatever I needed. With each of your comments and discussions with you, I was able to shape and improve my skills further, which lead me to successfully finish my PhD.

To you both, I have no words and cannot thank you enough for giving me the opportunity of pursuing my PhD with you.

Dears Thomas, Begül, and Oscar, you have been my mentors, who boosted my work and skills that have brought me to this point. Thank you for your patience in the lab, for your ideas, and for helping me with my horrible writing at the beginning. You helped to get through the most difficult times of my PhD.

And Lauren, my teammate, my close companion on this trip. With just a few months difference, we began our trip together, and now we finalize it together. We have worked, traveled, discussed, suffered, enjoyed, and had a roller-coaster of emotions throughout these years, where you have always supported me. I

will not forget those endless weekends in the lab, our sushi dinners, our trips, both for work and for fun, and our eternal and profound discussions about life itself. We even had time to share the same office. All those things contributed to shape this unforgettable experience.

I would like to extend my gratitude to the jury members, who thoroughly reviewed my text allowing me to improve it further. To all my co-authors, who made, in no small part, my papers possible. To all generations of office mates, with whom I had so much fun, and who made my office a place to look forward to going every day. Thank you.

How to forget my Barracks crew, and all our adventures. Summer schools, Sofia, COSIC weekends, camping, Spartan Race, New Castle, skying trips, Metafore, chiliwilis, and many others. Thanks to you, these years seemed like an eternal summer, eternal holidays. You have made these years wonderful. To the ping-pong team, cause all those moments contributed to make this PhD easier. To Péla, and all the COSICs which names I cannot fit entirely here. To my BJJ team, for keeping me in good shape both physically and mentally.

Quiero dar las gracias a mis padres, a los que estaré eternamente agradecido, porque sin su apoyo no estaría donde estoy. Porque nunca han dudado de mí, y me han dado alas para llegar a donde yo quisiera. A tí, Dani, por tu confianza ciega, tu apoyo infinito, y tu amor incondicional. Porque me escuchas y me entiendes, sean cuales sean mis preocupaciones. Ojalá el tiempo se detuviera en esos días juntos en Lovaina con nuestros entggggenacos. Os quiero.

A Jorge, por esos skypes de "ssseda" los domingos a la tarde en la lavanderia, que me mantuvieron a flote en los momentos mas duros. A Andrea, Borja, Eloi, Sayus, y Unai, los más machines, por esas visitas y charletas eternas que tanto nos encantan, y que tanto me ayudan.

Finalment, per acabar però no menys important a tu, Júlia. La meva musa, la que em guia, em complementa i em fa més fort. Vull agrair-te de tot cor que vinguessis fins a Leuven, perquè sense tu al meu costat no hagués arribat mai on estic. Per tots aquests "papers" que t'has llegit, i totes les presentacions que has escoltat, que per molt complicats o tècnics que fossin sempre has sabut ajudar-me perquè fossin millors. Per escoltar totes les meves idees, i donar-me suport inclús en les més boges, mantenint-me sempre amb els peus a la terra. Perquè no et canses d'empènyer quan les meves forces flaquegen, i , per moltes queixes que et posi, sempre et mantens forta. Finalment, perquè m'ajudes a mantenir els valors que més m'importen, i fas que tregui la millor versió de mi mateix. T'estimo.

Victor Arribas

May 2020

Abstract

The era of the Internet of Things (IoT) is rapidly giving way to the Internet of Everything (IoE) era, in which everything is connected. Hundreds of new different smart devices are devised within no time, where wearables, smart homes, or even smart bikes highly interconnect our lives. These communications generate a tremendous amount of data, which is extremely sensitive. The need for privacy, anonymity, and security, among others, arises. Cryptographic algorithms can provide all these capabilities, which are then implemented in these smart devices, turning them in what we call cryptographic devices.

Cryptographic devices are usually within reach of malicious users, which allows them to perform physical attacks to retrieve secret data or corrupt secret computations. A traditional adversary wanting to compromise a cryptographic algorithm was assumed to have only access to several inputs and outputs of the algorithm. Physical attacks imply additional information, making it easier for an attacker to by-pass the security of the device. These attacks are commonly divided into two categories, namely, passive and active attacks, or Side-Channel Analysis (SCA) and Fault Analysis (FA), respectively. The most widely investigated countermeasure against SCA is masking, while for FA, redundancy or Concurrent Error Detection (CED) schemes are the most utilized. Even though it depends on the application, these countermeasures are commonly researched independently. Nevertheless, given the accessibility of these devices, an attacker could just perform Fault Analysis in a cryptographic device protected against Side-Channel Analysis, and vice-versa. Thus, designers should not neglect one countermeasure or the other but implement them jointly.

The environment in which cryptographic devices execute is usually highly sensitive. The correct implementation of the security components is critical since failure could lead to substantial losses, ultimately affecting an individual's integrity. Thus, the verification of the security provided and the correct behavior of these countermeasures is crucial. Experimental security evaluations require special equipment, a considerable amount of time, human intervention, and

extensive technical knowledge. Taking into account the number of cryptographic devices being deployed, it is impossible to evaluate the security of each of them experimentally. Therefore, the need to automate this process to speed up the security assessment of a device.

In this thesis, we tackle both the design and the verification of combined countermeasures resistant against SCA and FA in hardware. The two tasks were performed in tandem, each providing feedback to the other. On the one hand, the acquired knowledge on designing implementations improved the quality of our verification procedures. On the other hand, we could use the verification tools to double-check the correctness of our implemented schemes. The two areas of study are developed following the same procedure to achieve the goal of design and verification of combined countermeasures. We start by researching on SCA countermeasures, and then continue with the investigations on FA, to reach the goal of designing and correctly verifying combined countermeasures.

In the first part of the thesis, we include several implementations of (physically) unprotected cryptographic algorithms. Then, we study different masking countermeasures and their trade-offs when implemented in hardware. Additionally, we propose a methodology to secure low latency or unrolled hardware implementations. Subsequently, we analyze the resistance of SCA countermeasures against a particular attack, which we consider a hybrid between passive and active. Finally, we propose two different schemes resistant to combined attacks.

In the second part of the thesis, we elaborate on the verification procedures proposed and the set of tools implemented. First, we implement a verification tool to check the correct implementation of a particular masking scheme known as Threshold Implementations (TI). Subsequently, we propose a fault evaluation strategy, which includes a new fault model and consolidates fault models previously proposed in the literature. Both tools achieve remarkable performance in practice, and were able to find several algorithmic and implementation flaws on implementations previously published in the literature.

Beknopte samenvatting

Het tijdperk van het Internet of Things (IoT) maakt snel plaats voor het Internet of Everything (IoE) tijdperk, waar alles verbonden is. Honderden nieuwe en verschillende slimme apparaten zijn in een mum van tijd bedacht en wearables, slimme huizen of zelfs slimme fietsen brengen alle aspecten van ons leven in verbinding. Deze communicatie genereert een enorme hoeveelheid data, die uiterst gevoelig is. De behoefte aan privacy, anonimiteit en veiligheid onder andere groeit enorm. Cryptografische algoritmen kunnen al deze mogelijkheden bieden, die vervolgens op deze slimme apparaten worden geïmplementeerd, waardoor ze worden omgezet in wat we cryptografische apparaten noemen.

Cryptografische apparaten zijn meestal binnen het bereik van kwaadaardige gebruikers, die fysieke aanvallen kunnen uitvoeren om geheime gegevens te achterhalen of de berekeningen te verknoeien. Een traditionele aanvaller die een cryptografisch algoritme wilde analyseren, werd lang verondersteld alleen kennis van de ingangen en uitgangen van het algoritme te hebben. Fysieke aanvallen leiden tot aanvullende informatie, waardoor het voor een aanvaller gemakkelijker wordt om de beveiliging van het apparaat te omzeilen. Deze aanvallen worden meestal onderverdeeld in twee categorieën, namelijk passieve en actieve aanvallen, respectievelijk nevenkanaal analyse (NKA) en fout analyse (FA). De meest onderzochte tegenmaatregel tegen NKA is maskeren, terwijl voor FA, redundantie of Gelijktijdige Error Detectie (GED) schema's het meest worden gebruikt. Gewoonlijk worden deze tegenmaatregelen onafhankelijk onderzocht. De juiste tegenmaatregel om te implementeren, hangt uiteraard af van de toepassing. Gezien het feit dat beide aanvallen steeds eenvoudiger worden om uit te voeren, kan een aanvaller fout analyse uitvoeren op een cryptografisch apparaat dat is beveiligd tegen NKA en omgekeerd. Daarom moeten ontwerpers de ene of de andere tegenmaatregel niet verwaarlozen, maar gezamenlijk implementeren.

De omgeving waarin cryptografische apparaten werken, is meestal zeer gevoelig. De juiste implementatie van de beveiligingscomponenten is van cruciaal

belang, aangezien een storing tot aanzienlijke verliezen kan leiden, met uiteindelijk menselijke schade tot gevolg. Daarom is de verificatie van de aangeboden beveiliging en het juiste gedrag van deze tegenmaatregelen cruciaal. Experimentele veiligheidsevaluaties vereisen speciale apparatuur, een aanzienlijke hoeveelheid tijd, menselijke tussenkomst en uitgebreide technische kennis. Rekening houdend met het aantal cryptografische apparaten dat wordt ingezet, is het onmogelijk om de beveiliging van elk van hen experimenteel te evalueren. Daarom moeten we dit proces automatiseren om de beveiligingsbeoordeling van een apparaat te versnellen.

In dit proefschrift behandelen we zowel het ontwerp als de verificatie van gecombineerde tegenmaatregelen die bestand zijn tegen NKA en FA in hardware. De twee taken werden tegelijk uitgevoerd en elk gaf feedback aan de andere. Enerzijds, verbeterde de verworven kennis over het ontwerp van implementaties de kwaliteit van de verificatieprocedures. Anderzijds konden de verificatiehulpmiddelen de juistheid van de geïmplementeerde schema's controleren. De twee studiegebieden werden ontwikkeld volgens dezelfde procedure om tot het ontwerp en de verificatie van gecombineerde tegenmaatregelen te komen. We zijn begonnen met onderzoek naar SCA-tegenmaatregelen. We hebben vervolgens onderzoek over FA toegevoegd om het doel te bereiken van gecombineerde tegenmaatregelen te ontwerpen en correct te verifiëren.

In het eerste deel van het proefschrift nemen we verschillende implementaties van (fysiek) onbeschermde cryptografische algoritmen op. Vervolgens bestuderen we verschillende maskerende tegenmaatregelen en hun afwegingen bij implementatie in hardware. Daarnaast stellen we een methode voor om hoge snelheid of "uitgerolde" hardware-implementaties te beveiligen. Vervolgens analyseren we de weerstand van SCA-tegenmaatregelen tegen een bepaalde aanval, die we als een hybride tussen passief en actief beschouwen. Ten slotte stellen we twee verschillende tegenmaatregelen voor die bestand zijn tegen gecombineerde aanvallen.

In het tweede deel van het proefschrift gaan we dieper in op de voorgestelde verificatieprocedures en de geïmplementeerde set hulpmiddelen. In eerste instantie, presenteren we een verificatietool om de juiste implementatie van een bepaald maskerschema, Threshold Implementaties (TI), te controleren. Vervolgens stellen we een foutevaluatiestrategie voor met onder andere een nieuw foutmodel en consolideren we eerder in de literatuur voorgestelde foutmodellen. Beide tools presteren goed in de en konden verschillende algoritmische en implementatiestromen vinden in implementaties die eerder in de literatuur waren gepubliceerd.

List of Abbreviations

AES Advanced Encryption Standard

ATPG Automatic Test Pattern Generation

CAESAR Competition for Authenticated Encryption: Security, Applicability and Robustness

CAPA Countermeasure Against Physical Attacks

CASCADE Computer Aided Side Channel Analysis Design Environment

CED Concurrent Error Detection

CFD Cryptographic Fault Diagnosis

CHES Cryptographic Hardware and Embedded Systems

CMS Consolidated Masking Scheme (CMS)

CPA Correlation Power Analysis

DFA Differential Fault Analysis

DOM Domain Oriented Masking (DOM)

DOM Difference-of-Means

DPA Differential Power Analysis

DSD Digital System Design

EDA Electronic Design Automation

EDC Error Detecting Code

FA Fault Analysis

FDTC Fault Diagnosis and Tolerance in Cryptograhy

FIB Focused Ion Beam

FPGA Field Programmable Gate Array

FPL Field Programmable Logic

FS Fault Sensitivity

FSA Fault Sensitivity Analysis

HOST International Symposium on Hardware Oriented Security and Trust

IC Integrated Circuit

ICECS International Conference on Electronics, Circuits, and Systems

IoT Internet of Things

IoE Internet of Everything

MICPRO Microprocessors and Microsystems

MPC Multi-Party Computation

RSA Rivest, Shamir and Adleman

SA0 Stuck-at-0

SA1 Stuck-at-1

SCA Side-Channel Analysis

SEA Safe-Error Attack

SET Single Event Transient

SEU Single Event Upset

SHA-3 Secure Hash Function 3

SIFA Statistical Ineffective Fault Attack

SPA Simple Power Analysis

TA Template Attack

TCHES Transactions on Cryptographic Hardware and Embedded Systems

TI Threshold Implementations

TVLA Test Vector Leakage Assessment

VerMI Verification tool for Masked Implementations

VerFI Verification tool for Fault Injection

VerMFi Verification tool for Masked implementations and Fault injection

WDDL Wave Dynamic Differential Logic

Contents

Abstract iii

Beknopte samenvatting v

List of Abbreviations vii

List of Symbols ix

Contents ix

List of Figures xv

List of Tables xix

1 Introduction 1

1.1 Threats and Adversarial Model 2

1.1.1 Adversarial Model 3

1.1.2 Threats 4

1.1.3 Countermeasures 5

1.2 Motivation 6

1.3 Research Questions 9

1.4 Thesis Roadmap 11

2 Physical Attacks **13**

2.1 Notation 14

2.2 Side-Channel Analysis 14

2.2.1 SPA & DPA 15

2.2.2 Countermeasures 16

2.2.3 Evaluation 16

2.3 Fault Attacks 17

2.3.1 Differential Fault Analysis 17

2.3.2 Countermeasures 18

2.3.3 Evaluation 18

2.4 Adversarial Models 19

2.4.1 Side-Channel Analysis models 19

2.4.2 Fault Analysis models 20

2.5 Actively secure MPC 20

2.6 Boolean Masking 21

2.6.1 ISW 21

2.6.2 Threshold Implementations 21

2.6.3 Consolidated Masking Schemes (CMS) 23

2.6.4 Domain Oriented Masking (DOM) 24

2.7 Logic Simulation and Reliability Testing 25

2.7.1 Logic Simulation 25

2.7.2 Integrated Circuits Testing 26

2.8 Security Evaluation and Performance Results 28

2.8.1 TVLA 28

2.8.2 Setup 29

2.8.3 Results 29

2.9 Conclusion 29

3 Implementations and SCA countermeasures for the Keccak family 31

3.1 KECCAK family algorithms 32

3.1.1 KECCAK permutations 32

3.1.2 KETJE 34

3.1.3 SHA-3 40

3.2 Beyond the Limits: SHA-3 in just 49 Slices 41

3.2.1 Related work 42

3.2.2 Row-Based Serial Architectures 43

3.2.3 Area-Throughput Efficiency 47

3.2.4 Results 49

3.3 Towards Secure Hardware 51

3.3.1 Input dependencies 52

3.3.2 Changing of the Guards 54

3.4 Guards in Action: First-Order SCA Secure Implementations of Ketje 55

3.4.1 Secured Implementations 55

3.4.2 Evaluation 58

3.4.3 Results 60

3.5 Rhythmic KECCAK: SCA Security and Low Latency in HW . . 62

3.5.1 Securing Keccak with $d+1$ masking 62

3.5.2 Low Latency Masking 69

3.5.3 Speeding Up KECCAK Implementations 71

3.6 Conclusions 76

4 Glitch-Resistant Masking Schemes as Countermeasure Against Fault Sensitivity Analysis 77

4.1 Fault Sensitivity Analysis 78

4.2 Related Work 79

4.3 The power of SCA glitch-resistance against Fault Sensitivity Analysis (FSA) . . . 79
4.3.1 Assumptions . . . 79
4.3.2 On Non-Completeness . . . 80
4.3.3 On Uniformity . . . 83
4.4 Experiments . . . 84
4.5 Results . . . 85
4.6 Discussions . . . 89
4.6.1 The Roche and Prouff Scheme . . . 89
4.6.2 ASIC vs FPGA . . . 90
4.6.3 Reset Value . . . 91
4.7 Conclusions . . . 91

5 Design of SCA and FA Combined Countermeasures 93
5.1 The Tile-Probe-and-Fault Model . . . 94
5.2 CAPA . . . 98
5.2.1 Evaluation Stage . . . 99
5.2.2 Preprocessing Stage . . . 103
5.2.3 Relation Verification of Auxiliary Data . . . 104
5.2.4 Hardware Implementations . . . 106
5.2.5 Cost Analysis and Scalability . . . 111
5.3 M&M . . . 112
5.3.1 Basic M&M Building Blocks . . . 115
5.3.2 Building circuits with M&M . . . 116
5.3.3 Infective Computation . . . 122
5.3.4 AES Case Study . . . 125
5.4 Security Claims and Discussion . . . 134
5.4.1 Security Analysis CAPA . . . 134

5.4.2 Security Analysis M&M 136
5.4.3 Attacks 138
5.4.4 On the MAC Tags 140
5.4.5 Differences between CAPA and SPDZ 142
5.5 Conclusion 142

6 Verification of SCA and FA countermeasures - VerMFi 145
6.1 Logic and Fault Simulators 149
6.1.1 Logic Simulator 149
6.1.2 Fault Simulator 150
6.2 VerMI 153
6.2.1 Header 154
6.2.2 Non-completeness 154
6.2.3 Uniformity 156
6.2.4 Sequential circuits 157
6.3 VerFI 161
6.3.1 Cryptographic Fault Diagnosis 161
6.3.2 Framework 165
6.3.3 Faults configuration 166
6.3.4 Evaluation 167
6.3.5 Statistical Ineffective Fault Attack 167
6.3.6 Infective Countermeasures 168
6.4 Case Studies 169
6.4.1 DOM Keccak 169
6.4.2 Masked Keccak S-boxes 169
6.4.3 CAPA and M&M 170
6.4.4 ParTI 171

6.4.5 Impeccable Circuits 175
6.5 Results 176
6.5.1 Evaluation 176
6.5.2 Performance 179
6.6 Conclusions 182

7 Conclusion **183**
7.1 Summary and conclusions 183
7.2 Future work 187

A SCA Keccak miscellaneous **191**
A.1 Keccak Round Steps 191
A.2 Second-order Masking 191
A.2.1 Threshold Implementation (TI) 191
A.2.2 Consolidated Masking Scheme (CMS) 192
A.3 Second-Order Non-completeness Failure in Round-Based DOM Implementation 193
A.4 Double round sharings 195
A.4.1 $5 \rightarrow 10$ 195
A.4.2 $10 \rightarrow 5$ 197
A.4.3 First layer $6 \rightarrow 6$ 198
A.4.4 Second layer $6 \rightarrow 6$ 199

Bibliography **201**

List of Publications **221**

List of Figures

1.1 Contribution progress throughout the thesis. 9

3.1 KECCAK-p [200] round-based implementation. 33

3.2 KECCAK-p^* round function extending the original KECCAK-p 34

3.3 The MONKEYDUPLEX construction [35] 35

3.4 Architecture for the MONKEYDUPLEX implementation. Black datapath for the unprotected version, additional red dotted for the protected. 36

3.5 The MONKEYWRAP construction [35] 37

3.6 The MONKEYWRAP implementation including encryption&decryption 39

3.7 Sponge construction. Taken from [2]. 40

3.8 Row-based implementation. 43

3.9 SHA-3 lightweight architectures framework. 44

3.10 SHA-3 lightweight architectures control FSM. 45

3.11 SHA3-TPA architecture framework. 48

3.12 SHA-3 synthesis results. 50

3.13 Independent and dependent inputs CMS/DOM sharing example 52

3.14 "Changing of the Guards" structure for three adjacent S-Boxes from a three shares secure implementation [69] 54

3.15 Welch t-test evaluation of full KECCAK-f. Top to bottom: power trace, first- and second-order masks off test, and first- and second-order masks on test 59

3.16 Welch t-test evaluation of full KETJE encryption. Top to bottom: first- and second-order masks off test, and first- and second-order masks on test 60

3.17 Datapath of protected (blue) and unprotected round-based Keccak implementation with and without χ-compression respectively. 63

3.18 Tracing back output bit $C^{\chi}[4, 1, 0]$. 65

3.19 TVLA on our first-order DOM-KECCAK implementation. Left: Masks off with 20k measurements. Right: masks on with 55M measurements. From top to bottom: first-, second-, and third-order analysis respectively. 66

3.20 One round structure of the naive fix, with refreshing ($\mathcal{R}$) and compression ($\mathcal{C}$) before ι. 67

3.21 One round structure of NC-KECCAK, with the state register after the linear permutations and ι before the second layer of registers. 68

3.22 TVLA on our first (left) and second-order (right) NC-KECCAK implementations over 55M traces. From top to bottom: first-, second-, and third-order analysis respectively. 68

3.23 Quartic operation with (left) first-order sharing and (right) second-order sharing for the first round/layer of operations. . . 70

3.24 TVLA on first-order secure double round KECCAK implementation. Left: Masks off. Right: masks on. From top to bottom: first-, second- and third- order analysis. 75

4.1 General structure of a Threshold Implementation 81

4.2 First share combinational circuit from the 3-share AND gate en Eqn. (4.1) 82

4.3 Circuit under attack, unprotected. 85

4.4 Circuit under attack, protected. 86

4.5 ASIC vs FPGA correlations for all profiles of the unprotected (top) and protected (bottom) PRESENT S-Box. 87

4.6 ASIC FS of the unprotected PRESENT S-Box for the highest correlation profile. 88

4.7 FSA attack on the unprotected (top), and TI (bottom) implementation of the PRESENT S-Box. 89

4.8 FSA attack on the unprotected (top), and TI (bottom) implementation of the KECCAK S-Box. 90

5.1 Partition of the integrated circuit area into tiles, implementing MPC "parties" 95

5.2 Non-specific leakage detection on the first 31 rounds of first-order KATAN. Left column: PRNG off (24K traces). Right column: PRNG on (100M traces). Rows (top to bottom): exemplary power trace; first-order t-test; second-order t-test 108

5.3 Non-specific leakage detection on the first 31 rounds of second-order KATAN. Left column: PRNG off (24K traces). Right column: PRNG on (100M traces). Rows (top to bottom): exemplary power trace; first-order t-test; second-order t-test; third-order t-test 108

5.4 AES S-Box pipeline 110

5.5 Infective Computation Scheme such as that of [131] 113

5.6 Our scheme 113

5.7 M&M non-linear operations: Obtaining the value and tag shares of $z = xy$(left) and $z = x^2$(right) 116

5.8 Obtaining the value and tag shares of $z = x^5$ 118

5.9 Inversion pipeline. (Register stages are depicted by red dotted vertical lines.) 119

5.10 M&M inversion: Obtaining the value and tag shares of $z = x^{-1}$ 120

5.11 Ring and Additive Refreshing [65] 127

5.12 Non-specific t-test on second-order secure M&M AES implementation, V1. Left: PRNG off (24K traces); Right: PRNG on (100M traces). Rows (top to bottom): one exemplary power trace; first-order t-test; second-order t-test; third-order t-test. . . 131

5.13 Non-specific t-test on second-order secure M&M AES implementation, V2. Left: PRNG off (24K traces); Right: PRNG on (100M traces). Rows (top to bottom): one exemplary power trace; first-order t-test; second-order t-test; third-order t-test. . 132

5.14 Bivariate t-test on second-order secure M&M AES imlementation, V1 (left) and V2 (right). Below diagonal: PRNG off (20K traces); Above diagonal: PRNG on (50M traces). 132

5.15 Overview of the scheme 137

6.1 VerMFi's flow diagram. 146

6.2 Left to right: complete sensitive data set, third share (Ex. 6.1) non-complete, and third share (Ex. 6.1) complete. 155

6.3 Uniformity check example. 156

6.4 Top: sequential circuit. Bottom: combinational sub-circuits for first and second stages. 157

6.5 Injecting fault on a hierarchical module (H3) from the inputs in the gate-output fault model $\mathcal{M}_{\mathcal{O}}$ (left) and in the gate-input fault model $\mathcal{M}_{\mathcal{I}}$ (right). 163

6.6 Uniformity analysis on KECCAK S-boxes [31, 38, 69] respectively from top to bottom 170

6.7 Structure of LED protected with ParTI, taken from [168]. 171

6.8 Computation and unmasking of the error check vector in ParTI for first-order security, taken from [168]. 172

6.9 Implementation of round-based Midori for a small degree of redundancy, taken from [5]. 175

6.10 Non-completeness evaluation, time versus number of output sensitive bits. 180

6.11 Evolution of VerFI's execution time with respect to the number of toggles and the number of faults per injection. 181

A.1 KECCAK-f[200] state and steps [32]. 192

List of Tables

3.1 Four KETJE authenticated encryption schemes [35] 38

3.2 Post-Place and Route Results for SHA3-256. Results given for this and previous works for Virtex-5, -6 and -7, and Spartan-6. 49

3.3 Synthesis results for the entire KETJE family using the NanGate 45nm Open Cell Library 61

3.4 Synthesis results for the different designs 69

3.5 KECCAK-f[200] first-order secure unrolled implementations' performance results using NanGate 45nm library. Comparison with Parallel implementations from [38] and PARALLEL double clocked and PARALLEL pipelined from [99] 74

5.1 Area (GE) of 2-share KATAN-32 implementations with m MAC keys $\alpha[j] \in \mathbb{F}_q$ 107

5.2 Area (GE) of 3-share KATAN-32 implementations with m MAC keys $\alpha[j] \in \mathbb{F}_q$ 107

5.3 Areas for first- and second-order AES implementations with $m = 1$ in 2-NAND Gate Equivalents (GE) 110

5.4 The number of randomness in bytes for the initial sharing, shared multiplication and the sacrifice required for AES S-Box 111

5.5 Overview of the number of $\mathbb{F}_q$ multiplications (.), $\mathbb{F}_q$ additions (+) and linear operations in $GF(2)$ (L(.)) required to calculate all building blocks with $n_{\mathcal{T}}$ shares and m tags 111

5.6 Randomness Cost for the AES S-Box implementations 128

5.7 Cost of first- and second-order AES implementations 129

5.8 Area comparison for first-order secure implementations 130

5.9 Area comparison for second-order secure implementations 131

6.1 All undetected Two Bit Faults in each Nibble of ParTI-LED: Δ_J denotes the input fault of operation J. 173

6.2 Verification Tool benchmark 177

6.3 Performance evaluation of different fault and combined countermeasures from the literature, including fault model, number of simulations, inputs and faults, injections cycles, and coverage. . 178

"Journey before destnation"
— The Way of Kings

Chapter 1

Introduction

We live in the Internet of Things (IoT) era, in which everything is connected via smart devices. Some also refer to it as IoE or the Internet of Everything. These embedded devices are increasingly implementing security protocols and cryptographic algorithms or storing secret data. Due to the nature of these accessories, they are easily accessible to an attacker and susceptible to physical attacks. The attacker can tamper with the device to break the security or retrieve the secret data, even when a completely secure cryptographic algorithm is present. The attacker can bypass the security by using the power consumption of the electromagnetic emanations from the device, which are known as side-channels. Additionally, the attacker can actively alter the computation (by injecting faults) to break the security.

The era where security was only a military asset or just needed to preserve confidential state data is over. With the proliferation of laptops, smartphones, tablets, and other smart devices, there is a need for modern security to preserve the privacy of all individual users. Modern security employs *cryptography* to achieve the desired protection, which includes privacy, confidentiality, data integrity, or authentication, among many other security properties. Privacy is the capability of some users' data to remain unknown, giving them the ability to express themselves selectively. Confidentiality ensures that data remains confidential, that is, unavailable to an unauthorized public. Data integrity ensures that a message has not been altered from its original form when delivered to its recipient. Authentication provides the means to verify the identity of the person that sent a particular message. Cryptography provides the necessary assets to achieve these properties in the form of mathematical functions that map an input message to an output text using a cryptographic

key. This process is known as encryption, where the cryptographic algorithm remains known (publicly available), and the only unknown component is the (secret) key [119]. Cryptography is divided into two principal branches, namely *symmetric cryptography* and *public-key cryptography* or asymmetric cryptographic. The main difference is that the former one uses the same key for encryption and decryption, while the second one employs one key for encryption and a different one for decryption. The most important algorithms for respective branches are the Advanced Encryption Standard (AES) [71], and RSA (Rivest, Shamir and Adleman) [160].

A cryptographic algorithm is considered secure when it is computationally impractical to retrieve the key by simply trying different key instances to get the correct ciphertext given a plaintext. This blind key search is known as a brute force attack, and computationally infeasible means that a brute force attack does not work within a reasonable amount of time. State-of-the-art cryptographic algorithms would need hundreds of years to be broken with a brute force attack. *Cryptanalysis*, complementary to cryptography, studies methods on how to break the cryptographic algorithms. To break a cryptographic algorithm means finding a method (attack) that translates into a reduction of the keyspace provided by the algorithm in the original instance, such that a brute force attack is successful within a reasonable amount of time.

These cryptographic algorithms are developed and live in a theoretical realm, which are then brought to reality via implementation, typically in a personal computer, a microcontroller, or dedicated electronics. *Cryptographic devices* are electronic devices that implement cryptographic algorithms and that store cryptographic keys [133]. Nonetheless, the implementation of cryptographic algorithms brings a whole new set of affairs regarding practical security. Not only must the implemented algorithm be secure, but also the implementation itself. Now-a-days we have to consider the notion of breaking a cryptographic device, which means distilling the key of the device. The process of seeking to retrieve the secret key from a device is known as a physical attack.

1.1 Threats and Adversarial Model

In this section, we elaborate further on the physical attacks against a cryptographic device. We explain how the attackers are modeled, stating their capabilities at the highest level. The most widespread models in cryptography include the black-box, the gray-box, and the white-box models. Then, we present a classification of the different kinds of physical attacks, and finally, we introduce briefly possible countermeasures.

1.1.1 Adversarial Model

An adversary is an entity that aims to tamper with either a cryptographic algorithm or a cryptographic device in order to learn secret information. The adversarial model comprises the capabilities, assumptions, and limitations of an adversary, granting the security notions to a given algorithm.

The first adversarial model to be used in cryptography was the *black-box*. This model grants access to the adversary solely to plaintext-ciphertext pairs, which means that implementation characteristics are unknown to the attacker. Under this model, the analysis of the crypto system is implementation-independent, and making use of the concepts of cryptanalysis.

Currently, the *white-box* model is becoming increasingly popular for software implementations, due to the nature of the platforms where cryptographic algorithms are deployed. In this model, the attacker can fully manipulate the execution platform and even the software implementation. The attacker has also access to the memory bank during the execution of the algorithm or the capability to analyze the program's binary. To protect against this attacker, a program where the operations on secret data are mixed with random operations in such a way that they are both indistinguishable is often used. This strategy is known as code obfuscation.

Finally, the model that best encapsulates embedded software and hardware implementations security, and the one that we will be using throughout this work is the *gray-box* model. In this model, the adversaries have access to the device, from which they can query plaintext-ciphertext pairs, and record any side-channel. Additionally, they can tamper with it by injecting faults or making it function under extreme conditions (lower voltage, higher temperatures). This model enables multiple new physical attacks in addition to the mathematical ones using cryptanalysis. The first attack of this kind was performed by Kocher [123], where he exploited the fact that the timing of the operations was dependent on the secret data. The power consumption or the electromagnetic radiations of a device can also be exploited, leading to powerful attacks [124] for revealing the secret key of a cryptographic algorithm. Additionally, an attacker may inject faults in the device, to ease enormously cryptanalysis methods to the point where RSA or AES can be broken with a single faulty ciphertext pair [45, 143].

1.1.2 Threats

A large amount of different physical attacks are known to pose a severe menace to the security of cryptographic devices. These physical attacks vary considerably in terms of time, cost, expertise, and equipment needed. Below, we categorize these attacks according to different criteria.

The first, based on the attackers' degree of manipulation of the device, divides the attacks among passive or active:

- **Passive attacks:** In a passive attack, the device runs as stipulated by design, without the attacker affecting the computation. The attacker then obtains the secret key by merely *observing* the characteristics of the device, *e.g.*, power consumption, execution time, or electromagnetic emanations.
- **Active attacks:** In an active attack, the attacker interacts (tampers) with the device to produce abnormal behaviors in the execution. The attacker retrieves the key by exploiting unexpected operations. These attacks are also known as *Fault Analysis (FA)*.

The second criterion arranges the physical attacks depending on the design layer affected. Different methods and equipment are needed to affect particular elements of the design. Depending on the layer affected, we can distinguish among invasive, semi-invasive, and non-invasive attacks, all of them being either passive or active.

- **Invasive attacks:** Invasive attacks are the most potent kind of attacks. The attacker has no limitations on the capabilities and methods used to tamper with the cryptographic device in order to retrieve the secret key.

  Typically, this kind of attack starts by depackaging the device to get access to the lower-level layers. Within this category, the attack is considered passive if only a physical probe is used to observe the data signals. On the other hand, it is treated as active if the computation is altered by affecting the data signals. Laser cutters or focused ion beams are the standard tools used for invasive active attacks. Even though this attack is extremely potent, it also requires remarkably expensive equipment.
- **Semi-Invasive attacks:** Like the previous brand of attacks, semi-invasive attacks also require depackaging the device. However, they do not need direct electrical contact to the chip; the passivation layer remains untouched.

  A passive semi-invasive attack typically consists in reading out the content of the memory cells without actually probing or making contact with the

circuit. An active semi-invasive attack objective is also to inject faults. This attack is possible using for example X-rays, electromagnetic fields, or light. These attacks use comparably less expensive equipment than the previous attacks, but they are still tedious, given that depackaging is still needed.

- **Non-Invasive attacks:** In this variety of attacks, the device is attacked without carrying any alteration. Only accessible interfaces are manipulated. The device is not permanently modified, which means no trace of the attack is left. Additionally, the conventional equipment for these attacks is rather inexpensive, which makes them a serious threat to the security of cryptographic devices.

  The passive attacks within this category are commonly known as *Side-Channel Analysis (SCA)*, arguably the attacks that have attracted the most interest in the literature from the last twenty years since the seminal paper of Kocher [123]. The typical exploited side-channels are execution time, power consumption, or electromagnetic emanations. By measuring these features, the attacker can recover the secret key. Active attacks within this category are performed via clock or power glitches, or by modifying the recommended temperature.

1.1.3 Countermeasures

Different countermeasures implemented in a cryptographic device can be utilized to thwart physical attacks. Implementing physical countermeasures always entails a higher cost in the design. Traditionally, a designer of embedded devices has to take into account trade-offs within three dimensions, namely, circuit complexity, throughput, and energy. Optimizing one category often means a penalty in one or the other two dimensions. For the design of cryptographic devices, a new dimension has to be taken into account: security. Security always comes with an extra cost, normally penalizing the performance of the three former dimensions. Currently, the challenge is to optimize the original three metrics while including a certain level of security. Trivially, the higher the level of security needed, the higher the extra costs in the implementation will be.

We sort the possible countermeasures according to the design layer where they are implemented. They can appear at the protocol, the algorithmic, the logic, or the physical level.

- **Protocol level:** The countermeasure is inherent in the protocol definition of the algorithm or the cipher. In the case of passive attacks, Leakage-resilient Cryptography [80] defines primitives that remain secure even if a

limited amount of arbitrary information on the internal state is leaked during computation. An example of leakage-resilient cryptography is re-keying protocols, where the key is changed often to prevent the attacker from acquiring enough traces for a successful attack. On the other hand, tweakable ciphers [26] have been proposed as cryptographic primitives resistant against fault attacks.

- **Algorithmic level:** Algorithmic countermeasures are additional theoretical measurements added in a cipher to provide physical security. This kind of countermeasures include masking [177] or hiding [133] as countermeasures against passive attacks, which aim to decouple the side-channel leakages from the secret data. As a countermeasure against fault attacks, Error Detecting Codes (EDCs) are typically used, which add some optimized redundancy to the circuit.
- **Logic level:** Logic level countermeasures are typically implemented directly on the netlist or the standard cells. Within this group, we can find Wave Dynamic Differential Logic (WDDL) [175] as a countermeasure against SCA. The idea behind this method is to make the cell's internal switching constant so that the power consumption of the logic cell is independent of the computed value.
- **Physical level:** Finally, a countermeasure can be implemented at the physical level, where different kinds of sensors or shields are used to detect the presence of a probe or prevent the injection of malicious faults.

Other methods to prevent physical attacks concentrate on hampering the data analysis (processing) phase of the attack. Strategies that have been widely used against Side-Channel Analysis attacks include the use of clock jitter, randomized order of operations execution, or the insertion of dummy operations, to produce substantial misalignment in the traces. Fault attacks are usually performed using pairs of legitimate and faulty ciphertexts to retrieve the secret key. Thus, a countermeasure that makes it impossible for a user to acquire more than a particular maximum number of queries could help to hinder these attacks.

1.2 Motivation

The brisk growth of cryptographic devices, in addition to all different possible physical attacks, makes the development of physical countermeasures a pressing concern. Furthermore, the equipment to accomplish these attacks and the attacks themselves are rapidly evolving, becoming cheaper and easier to reproduce, respectively. This progression means that the design

of countermeasures has to evolve accordingly, to have new and more efficient methodologies to ensure the security of cryptographic devices.

We find cryptographic devices in an abundant number of applications, ranging from smartcards, secure wallets for valuable assets, RFID tags for identification, pacemakers, or autonomous cars, amongst others. Conventional processors, present in all these applications, are strikingly vulnerable at the hardware level to multiple physical attacks, such as the ones known as Meltdown [130], and Spectre [122]. Moreover, even hardware secure enclaves providing roots of trust for higher-level applications, as it is the Intel SGX chip, have also been broken [50]. These kinds of attacks are rapidly proliferating, due to the more profound understanding of the nature of physical attacks, breaking innumerable security boundaries [51, 55, 169, 178] (all of them published this year). Most of these vulnerabilities come from the underlying hardware, which does not include any physical countermeasures. Millions of chips are already produced and commercialized with these vulnerabilities. Since the hardware cannot be updated, these weaknesses are mitigated with software or microarchitectural patches, but these fixes do not solve the core problem.

Surely, all these chips deployed into the market with these vulnerabilities were functionally verified at every stage of the design process and tested to ensure the correct behavior. In most of the cases, the testing procedure defines the final price of the device, which suggests how critical this process is. However, their security against physical attacks was most probably not verified. With the proliferation of all these attacks, verifying the security level of an implementation is as critical as it is checking for correct demeanor. Now, we must design with functionality and security in mind, assessing both of them. Similarly to behavioral verification, the security evaluation must be performed at every phase of the design flow.

As we have seen, it is indispensable to include physical countermeasures in cryptographic devices, and verify them accordingly. The countermeasure strength should go in line with the application and the level of security needed, to optimize the extra costs that this addition entails. Thus, investigations in new and efficient countermeasures are needed, as well as metrics and efficient methods for verification.

At the start of this research, the state of the art in SCA countermeasures focused on masking [40, 98, 108, 147, 149, 152, 157] as the preferred defense mechanism against SCA, which is provably secure, yet useful in practice. Ishai *et al.* [108] proposed the first secure masking scheme for any arbitrary order of attack. Nevertheless, they did not take into account the non-linear behavior of a hardware implementation (signals glitches), so their countermeasure was not suitable for this kind of implementations. The first work to cope

with glitches is the manuscript by Nikova *et al.* [147], known as Threshold Implementations (TI), which proposed a first-order SCA secure mechanism for hardware implementations. A subsequent article by Bilgin *et al.* [40] enhanced this masking scheme to provide security for higher orders. The following papers [98, 157] derive from the former sharing schemes to handle multi-variate attacks, and reduce the circuit complexity. From this studies, we can recognize two tracks in the masking schemes. On the one hand, TI-like techniques [40, 149], also known as $td + 1$ schemes, which, at first sight, would use more resources, but less latency, and less randomness. On the other hand, CMS/DOM-like approaches [98, 157], also known as $d + 1$ schemes, which, at first sight, would need more latency and extra randomness, but less circuit complexity.

In contrast to masking, the countermeasures for fault analysis were mostly heuristic and lacked a formal background. There are two principal countermeasures to protect against fault injections, as noted in [131]: The first, *Detection*, checks whether the algorithm was faulted during the execution by using either area or time redundancy (*e.g.* duplication [16], Concurrent Error Detection (CED) [30, 115, 116]). The problem with duplication is that it does not provide security when faults are duplicated as well. Even with error-detecting codes, a powerful attacker can avoid detection if the injected faults result in compelling codewords. The second approach, *Infection*, prevents an adversary from extracting secret information from a faulty ciphertext by ensuring that any induced fault results in a garbage output [93]. These schemes alleviate the need for a checking mechanism. However, they cannot give an honest user any indication of whether or not an execution was faulty.

In the past years, most of the studies about Side-Channel Analysis and Fault Analysis countermeasures have been carried independently. The research direction of combined countermeasures - that is, countermeasures against both SCA and FA - is quite young and experimental. A conventional methodology was to superpose two techniques that separately resist one family of attacks. Examples of schemes that combine masking against SCA with redundancy against FA are ParTI [168] and Private Circuits II [64, 107]. These countermeasures naturally inherit the drawbacks of redundancy, *i.e.*, they are vulnerable against the injection of *smart* undetectable faults. Moreover, implementing a checking mechanism that does not reveal sensitive information under combined attacks is a difficult task. A combination of duplication and infection is explored in [131], but this scheme was broken in [22].

Within the field of security verification, only the tool by Barthe *et al.* [18] to formally verify masked software implementations, or the Test Vector Leakage Assessment (TVLA) [25] methodology were publicly accessible. No automated procedures were available for the verification of Fault Analysis, and surely, neither strategies for the evaluation of combined countermeasures.

1.3 Research Questions

As mentioned before, research on combined countermeasures against both SCA and FA is very scarce. This thesis targets multiple open problems related to this topic, both on the design and the verification of combined countermeasures. The work on this thesis has been carried following a "bottoms-up" approach, building progressively on our research to get to the final goal, as illustrated in Fig. 1.1. As a foundation for the subsequent contributions, we start with the design and implementation of several unprotected ciphers, where we explore different design trade-offs, in particular focusing on area and speed performance. Then we add side-channel resistance capabilities to these implementations, studying different concepts to devise robust SCA countermeasures. Subsequently, we study how previously developed SCA countermeasures perform against a particular attack of a different nature, a hybrid attack, and analyze whether they can prevent it. Finally, we add the concepts of Fault Analysis to prior investigations, and propose new combined countermeasures. We explore different design trade-offs for the implementation of these countermeasures, including low area and low latency.

In addition to previous investigations, we research on automated procedures for the evaluation of both SCA and FA countermeasures. These studies were carried in parallel with the aforementioned contributions.

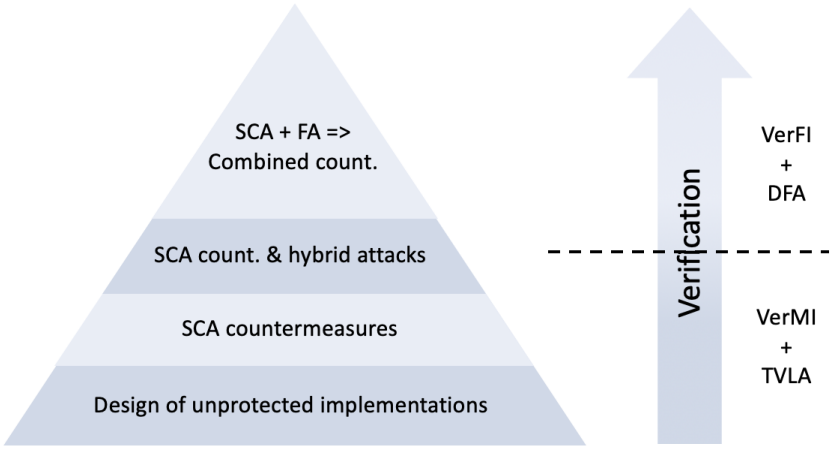


Figure 1.1: Contribution progress throughout the thesis.

Question 1. *What are the implications and trade-offs of choosing a $td+1$ vs. a $d+1$ masking scheme? How can we secure an unrolled low-latency implementation against Side-Channel Analysis?*

By implementing several protected designs of the SHA-3 underlying permutation [2], in Chapter 3, we investigate the differences in area, latency, and randomness between $td+1$ and $d+1$ sharing schemes. All the constructions of previous research focused on reducing the area overhead of the architecture when a countermeasure is implemented. Instead, we want to analyze what are the implications for the security mechanism applied when targeting lower latency implementations.

Question 2. *Can Side-Channel Analysis countermeasures prevent hybrid attacks?*

On our progress towards considering Fault Analysis and Side-Channel Analysis together, we look at a particular hybrid attack known as Fault Sensitivity Analysis (FSA) [129]. The attack injects a fault to tamper with the computation of the algorithm producing new side-channels, which then are exploited with SCA techniques to retrieve the secret. In Chapter 4, we check the performance of SCA countermeasures against this variety of attacks.

Question 3. *How can we design provably secure and practically feasible protection mechanisms against combined SCA and FA, and efficiently implement them?*

Masking derived originally from secret sharing [172] and Multi-Party Computation (MPC) techniques. Similarly, in Chapter 5, we devise a provable secure combined countermeasure by looking at the state-of-the-art MPC literature. Then, we optimize this architecture by relaxing the attacker model and the security warranties. As a must, when designing security, we need to evaluate the security of these implementations. Since no automated tools were at hand, we develop specific evaluation procedures for some cases.

Question 4. *How can physical countermeasures be efficiently and practically verified in an automated fashion? Which are the metrics that we have to check?*

In Chapter 6, we first build a tool to evaluate SCA countermeasures that checks the necessary conditions established by Threshold Implementations (TI) for the realization of secure implementations. Then, we broaden the tool's capabilities to support the evaluation of FA resistant implementations. We draft a logic and fault simulator as the basis to then construct the set of tools. It is

important to note that, even though we present this research question last, the investigations were pursued in parallel with the previous ones. We have carried the research of both design and verification of combined countermeasures in tandem, where the two tracks provided continuous mutual feedback. While crafting new countermeasures, one understands which concepts are important to look at in the verification process. The assertion of these concepts can then be automated in an evaluation procedure. At the same time, this automated verification ensures the correct application of the former principles.

1.4 Thesis Roadmap

This dissertation inspects critical concepts for the successful implementation of combined countermeasures against SCA and FA. Furthermore, it also investigates verification methodologies for these countermeasures. The layout of the thesis is derived from the research questions addressed in the respective order in the different chapters. Initially, Chapter 2 introduces the necessary concepts to understand subsequent chapters. We provide a more in-depth summary of SCA and FA, followed by the corresponding adversarial models. Then, we include more distinct concepts about active Multi-Party Computation, boolean masking, and logic simulation and reliability testing. Finally, we describe the methodology and the setup used for our experiments.

Chapter 3 encapsulates all the notions related solely to Side-Channel Analysis, where we present some unprotected implementations of state-of-the-art cryptographic primitives, illustrating different design trade-offs. Then, we propose several protected implementations, outlining various concepts critical to secure hardware countermeasures. Moreover, we exhibit design trade-offs for secured implementations, in particular targeting low latency unrolled implementations. The concepts discussed in this chapter are published in [8, 9, 11, 13], from which I am the main author.

Chapter 4 realizes the transition between chapters 3 and 5. We portray the basic concepts of Fault Sensitivity Analysis (FSA), and how it works. We provide theoretical and practical shreds of evidence on why TI-like and masking schemes secure in the presence of glitches protect against this sort of attack. The ideas from this chapter belong to the manuscript [10], produced in collaboration with Thomas De Cnudde and Danilo Šijačić, in which I devised the practical methods to prove the theory proposed, and performed the FPGA experiments.

Chapter 5 introduces all the concepts relative to the design of provable and practical combined countermeasures. It covers the theoretical aspects, and the strategies to implement these notions optimized for circuit complexity.

The concepts outlined in this chapter are publish in [74, 158]. This is a joint work with Begül Bilgin, Lauren De Meyer, Svetla Nikova, Ventzislav Nikov, Oscar Reparaz, Vincent Rijmen, and Nigel Smart, where my contributions include implementation and evaluation of the different designs, in addition to participating in the progress discussions.

Chapter 6 is dedicated to explaining in detail our tools-set for the verification of SCA and FA countermeasures. It is comprised of the explanation of both tools' features first, followed by the illustration of several case studies. The case studies presented include flaws the tools found in two architectures from the literature. The tools are published in [12, 14] and made available in [7].

Finally, we close this thesis with Chapter 7, which includes a summary of the conclusions of the thesis, as answers to the aforementioned research question, followed by open problems for future works.

Chapter 2

Physical Attacks

In this chapter, we describe the technical background relevant to our work. We provide high-level descriptions of the most critical physical attacks, namely Side-Channel Analysis (SCA) and Fault Analysis (FA). We give then a more detailed description of the most relevant attack techniques for both cases, Differential Power Analysis (DPA) [124] and Differential Fault Analysis (DFA) [37], respectively. Similarly, for both attacks, we describe the primary countermeasures studied in the literature: masking and hiding for SCA, and Concurrent Error Detection (CED) for FA, followed by the different evaluation procedures to verify the security of these countermeasures. Then, we outline the state-of-the-art for both adversarial models, which we will be using throughout the thesis.

Subsequently, we expound the concepts of actively secure Multi-Party Computation (MPC) and their use for SCA and FA countermeasures. We follow with further details about a specific family of SCA countermeasures, where we introduce the most recent methodologies presented in the literature. These methodologies are highly relevant to us since most of our contributions are based on them. Additionally, we present more detailed concepts about logic simulation, and reliability testing in digital circuits, which we later employ for the design of verification tools. Finally, together with the verification tools developed, we evaluate our designs with the method known as Test Vector Leakage Assessment (TVLA) [25], adopted and well recognized by the community for providing a thorough security assessment. We introduce said methodology together with the laboratory setup used for the security evaluations.

2.1 Notation

Throughout the entire manuscript, we work over a field $\mathbb{F}_q$ with characteristic 2, for example, $GF(2^k)$ for a given k, which is sufficient for the application to most symmetric ciphers. We use $\cdot$ and $+$ to describe multiplication and addition in $\mathbb{F}_q$, respectively. We use upper case letters for constants or matrices. The lower case letters x, y, z are reserved for variables. To denote that a variable x has been updated with a particular function, we use the notation x' to represent the new value (or output of the function). We use brackets to refer to a particular bit from a multiple-bit variable or a matrix. For example, $x[b]$ refers to bit b of x, while $S[b]$ refers to some bit or bits of the matrix S.

For protection against Side-Channel Analysis, we use solely Boolean masking as countermeasure, where we represent a secret split in different shares by $\vec{x} = (x_1, \ldots, x_s)$, such that $x = x_1 \oplus \ldots \oplus x_s$. The exact value of s is chosen depending on the desired security level. We denote s_{in} and s_{out} as the input and output number of shares, respectively, for a given operation. We use "$\cdot$" to denote a field multiplication in $\mathrm{GF}(2^k)$ and "$\odot$" for a shared (*i.e.* SCA protected) field multiplication in $\mathrm{GF}(2^k) : \vec{x} \odot \vec{y} = \vec{z} \Leftrightarrow x \cdot y = z$. Upright bold font is used for bit vectors $\mathbf{x} \in (\mathrm{GF}(2))^k$ and matrices $\mathbf{M} \in (\mathrm{GF}(2))^{k \times k}$. The degree of security is referred to as d, and the algebraic degree of an operation with t.

2.2 Side-Channel Analysis

Side-Channel Attacks (SCA) are based on observing the behavior of the device without making any changes on it or its working conditions (non-invasive). These attacks are frequently used due to their relatively low difficulty to be detected. They take advantage of the side-channels produced by a device to compromise the security of the cryptographic algorithm running. These side-channels come in the form of time, power consumption, electromagnetic emanations, or even sound. The most common attacks exploit the power consumption of a cryptographic device, which profit from the fact that the instantaneous power consumption depends on the data that is being calculated. The most prominent power analysis attacks include Simple Power Analysis (SPA) and Differential Power Analysis (DPA) [124]. Other notable attacks developed later include Template Attack (TA) [58], or Correlation Power Analysis (CPA) [47]. Furthermore, in the case where countermeasures are present, higher order attacks are employed. The best known attack is Higher-Order Differential Power Analysis (HO-DPA) [133]. These attacks are also referred to as d^{th}-order Side-Channel Analysis, which appoint a SCA attack of degree d.

2.2.1 SPA & DPA

The main difference between SPA and DPA lies in the number of traces used. SPA aims to reveal the key when only a small number of traces is provided. In the extreme case, one trace would suffice for an attacker to derive the key. Based on the shape of the power trace, the attacker could distinguish key-dependent patterns, *i.e.*, patterns depending on which bit of the key was used for the computation. In contrast, DPA attacks need considerably more traces than SPA attacks, but the main advantage is that no detailed information about the cryptographic device is needed. DPA analyzes the dependencies of power consumption with intermediate data at fixed moments. Additionally, DPA follows a divide-and-conquer strategy to recover the secret key, where instead of trying to recover 128 bits at once, the attack runs 16 times recovering one byte at a time. In the former case, there are 2^{128} candidates, while in the later, for each execution, there are just 2^8 candidates. Typically, DPA is mounted by following the next five steps:

1. **Choosing an intermediate:** the first step is to decide an intermediate computation value as a target for the attack. This intermediate should be the output of a function dependent on the key, preferably a non-linear function.

2. **Getting power consumption:** the second step in a DPA attack is to compute the power consumption of the design. The attacker should run several encryptions or decryptions to collect multiple traces.

3. **Calculating hypothetical intermediate value:** the next step is to make a hypothesis on the value of the intermediate for every possible candidate of the key part under attack (typically a byte or a nibble). Since the attacker knows the algorithm under attack and it is also the one inputting data to the device, it is possible to calculate all the targeted intermediate values for all the candidates of the key.

4. **Mapping hypothesis with power consumption:** the hypothesis on the intermediate value is then mapped to a power consumption value. A model of the power consumption is used to do the mapping, which could be, for example, to assume that the power consumption depends on the hamming-weight (HW) of the data.

5. **Hypothetical power consumption vs. Power traces:** in the last step of the attack, the correlation between the hypothetical power consumption and the actual power traces is analyzed. The candidate showing the highest correlation value reveals the key that has been used by the device.

2.2.2 Countermeasures

Multiple countermeasures have been proposed to thwart Side-Channel Analysis, which aim to make the power (or other sources of side-channel leakage) consumption independent of intermediate computations. The main countermeasures can be divided into two groups: hiding and masking. The idea is to either make the power consumption of the device as homogeneous as possible, where every operation requires similar power consumption, or to randomize the power consumption of the device. To mount an SCA attack, an attacker needs to collect a certain number of traces with a fixed key in order to retrieve its value. Hence, additionally to the previously mentioned countermeasures and as a general recipe, it is indispensable to change the cryptographic key as often as possible to prevent an attacker from gathering enough traces to perform a successful attack. Nevertheless, generating a fresh key is also not trivial, so we can not just generate a new key for every encryption.

With hiding, the power consumption can be randomized by executing the different operations at random points in time by, for example, randomly inserting dummy operations or by shuffling. The signal-to-noise ratio can be reduced by either lowering the signal, where all operations require the same amount of power for every input, or just by increasing the overall noise of the execution. Masking is the most researched countermeasure in the literature. It is based on secret sharing [172] and Multi-Party Computation (MPC) techniques, where the secret is split into pieces or "shares", and a minimum number of shares is required to reconstruct the secret. These schemes are often referred to as (s, n) threshold schemes, where s is the number of shares, and n is the minimum number of shares needed to recover the secret. The principal masking schemes proposed in the up-to-date literature use a full threshold scheme (s, s), *i.e.*, all shares are needed to reconstruct the secret. Each sensitive, i.e., key-dependent, data x is divided into s shares $(\mathbf{x} = (x_1, \ldots, x_s))$, such that $x = x_1 \perp \ldots \perp x_s$. Throughout this manuscript, we consider Boolean masking where $\perp$ is field addition denoted by $\oplus$ and full threshold sharing where all s shares need to be combined to derive x.

2.2.3 Evaluation

Evaluating the security provided by a given countermeasure is equally necessary. As functional verification is crucial to guarantee the correct functionality of a device, so it is the security validation to ensure the right protection. Security verification is a challenging task, generally needing a great deal of computational power and large amount of traces. It can be done at every stage of the design flow, *i.e.*, at the algorithm, the implementation, or the physical level,

where dedicated tools and specialized technical qualifications are needed. A considerable amount of work and resources are currently put into automating this verification process, resulting in multiple papers published in the literature in the past years [17,18,23,36,43,66,68,81,82,142,173]. There are tools meant to analyze software, while others are meant to analyze hardware implementations. Some of them are more formal and exhaustive, while others are more practical and precise.

2.3 Fault Attacks

Fault Attacks or Fault Analysis (FA) is the second crucial physical attack. The basic idea is to disturb the normal execution of a cryptographic algorithm by subjecting the cryptographic device to abnormal circumstances, like under-powering, over-heating, or careful illumination of the silicon. The result of a faulty computation can help an attacker to reveal information about the secret key. In the case of RSA or AES, a single faulty ciphertext pair makes key recovery possible [45, 143]. The most common fault attack is Differential Fault Analysis (DFA) [37]. Other distinguished fault attacks are Safe-Error Attack (SEA) [186], Fault Sensitivity Analysis (FSA) [129], and Statistical Ineffective Fault Attack (SIFA) [76]. In SEA, an attacker would fault an intermediate state of the computation setting it to a determined value (*e.g.*, bit set to '0'). If the computation does not change, the attacker would learn that the value of the targeted intermediate bit was '0'. In FSA, an attacker records the Fault Sensitivity (FS), or maximum critical path, of a particular input by injecting clock glitches. This Fault Sensitivity depends on the data due to the non-linear gates, and represents an exploitable side-channel. Statistical Ineffective Fault Attack (SIFA) exploits ineffective faults or faults not propagated to the output, which leads to a non-uniform distribution on intermediate values enabling a key-recovery attack.

2.3.1 Differential Fault Analysis

In a DFA [37] for a given plaintext and a specific fault model, the attacker can observe both the correct and faulty ciphertexts ct_c and ct_f, respectively. Given the pair of both outputs, the attacker guesses parts of the key and reverses the cipher computation up to the point of fault injection to check whether the difference computed corresponds to the considered fault model. Thereby an attacker can reduce the total number of key candidates with each (ct_c, ct_f) pair until very few or even one unique key candidate remains. Due to the diffusion properties of secure block ciphers, the attack is usually limited to the

penultimate or last round to avoid guessing the entire key, which would be computationally infeasible.

2.3.2 Countermeasures

The most straightforward countermeasures use simple redundancy, where the circuit is replicated n times to detect a total of $n-1$ faults. More sophisticated techniques include the use of Concurrent Error Detection (CED), which combines the original circuit A and a predictor, A', used to check the integrity of the original circuit. For increased performance, the predictor is a compressed mapping of the input, *e.g.*, a parity bit. Usually, Error Detecting Code (EDC) is the essential aspect of CEDs, typically used in data communication, where the sender encodes the message with some form of redundancy, allowing the receiver to detect (or even correct) a certain number of bits. Some fundamental concepts regarding EDC are [132]: a binary *linear* $[n,k]$*-code* $\mathcal{C}$ of length n and rank k conforms a vector subspace over $\mathbb{F}_2^n$ that forges a vector $x \in \mathbb{F}_2^k$ into codewords $c \in \mathcal{C}$. The *generator matrix* G maps a message x to its codeword by $x \cdot G = c \in \mathcal{C}$. A $k \times n$-matrix G is a generator matrix of an $[n,k]$-code $\mathcal{C}$ iff it is composed of k basis vectors of $\mathcal{C}$ of length n.

Definition 1. ***Minimum Distance.*** *The minimum distance of an* $[n,k,d]$*-code* $\mathcal{C}$ *is determined as follows:*

$$d = min\{HW(c_1 \oplus c_2) | c_1, c_2 \in \mathcal{C}, c_1 \neq c_2\},$$

where $HW : \mathbb{F}_2^n \mapsto \mathbb{N}$ *denotes the Hamming weight.*

Lemma 1. An $[n,k,d]$-code $\mathcal{C}$ can detect erroneous codewords $c' = c \oplus e$ iff $e \notin \mathcal{C}$ *i.e.*, e is not a valid codeword.

The detection capabilities of the linear code depend on the distance of the employed code. Hence, a linear code can detect up to $d-1$ faulty bits, which is the same as to say that $HW(e) \leq d-1$. The generator matrix G of a systematic linear code $\mathcal{C}$ is of the form $G = [I_k|P]$, where I_k refers to the identity matrix, and P refers to the predictor matrix. To obtain the check-bits of a message x, we can use the predictor matrix as $x' = x \cdot P$. This allows us to have a clear separation between the data and the predictor datapaths.

2.3.3 Evaluation

The evaluation of fault countermeasures is customarily carried with behavioral simulations before deployment, or by launching actual fault attacks on the

cryptographic device. Evaluation laboratories automate laser shoots along the entire surface of a chip, inject clock glitches at different times along the algorithm execution, or, for more advanced attacks, they use a Laser Cutter to target the digital logic. Some of them also use a Focused Ion Beam (FIB), but it is not so common since this is an enormously expensive attack.

Other than the real fault attacks on a cryptographic device, no free tools are currently known to (automatically) examine the security of countermeasures against fault-injection attacks where the adversary's boundary is defined. Apart from the difficulty of practical evaluations, it is highly challenging to limit the injected faults based on the assumed fault model. For this reason, only a reduced number of texts proposing countermeasures against Fault Analysis, evaluate their scheme [26, 30, 74]. To this end, the authors had to undergo an extremely tedious process of manually modifying the HDL code or the netlist to be able to perform fault injection in simulation, apart from the posterior data processing.

2.4 Adversarial Models

When designing algorithmic countermeasures, we need to make abstractions of reality, which comes with a model of how a real attacker behaves in a theoretical realm. Within this model, we have to describe the attacker's capabilities, which is known as the adversarial model. Trivially, the countermeasure will depend directly on the strength of the attacker. Protection against a stronger attacker requires stronger countermeasures, which entails more resources. In this section, we present the models that we will be using throughout the text for both SCA and FA.

2.4.1 Side-Channel Analysis models

The adversary model most used in the SCA literature was proposed by Ishai *et al.* [108], known as the d-probing model, where d refers to the number of probed variables in the circuit. The attacker can probe and read any intermediate variable. The adversary can get its value and the dependencies with previous variables, starting from the last layer of registers until the probed point, also know as glitch-extended probes [90, 157]. It has been shown in [79] that if a masked circuit achieves security under the d-probing model, then it also provides security against d^{th}-order Side-Channel Analysis under the independent leakage assumption.

2.4.2 Fault Analysis models

Besides the fact that most of the works in the literature provide only ad-hoc solutions for fault countermeasures, almost none of them consider a formal description of the adversary's capabilities. In contrast, ATP Security [91] provides a formal model and investigates algorithmic aspects for tamper-proof security. Later, Private Circuits II [107], and an extension of a glitch-free multiparty protocol [171] have defined a boundary for the adversary in order to guarantee the fault detection. The most common fault models described in the literature concerning fault injection in cryptographic algorithms [5, 16, 114, 150] concentrate on the number of faults allowed (limited to small number), when those faults should be injected and their duration (transient or permanent), and the type of the fault (stuck-at or bit-flip). Depending on how many faults are allowed, some fault models distinguish between single- and multiple-bit fault injections. Regarding the number of faults, countermeasures like duplication, EDCs [168], or Private Circuits II [107], limit the number of faults to be injected (usually per cycle) based on the level of redundancy or the distance of the underlying code. Concerning the timing of fault injection, the authors of Impeccable Circuits [5] introduce the Univariate ($\mathcal{M}$) and Multivariate ($\mathcal{M}^*$) concepts, in which stuck-at or Single Event Upset (SEU) faults can be injected in a single clock cycle or in multiple clock cycles respectively.

Usually, previous models either contemplate a uniform or a biased fault model: in the uniform model, the faults follow a uniform distribution, where each of them has the same probability of occurring. In contrast, a biased fault model is characterized by an increased chance of occurrence for particular faults (specific areas only).

2.5 Actively secure MPC

A passively secure Multi-Party Computation (MPC) protocol provides protection against a honest-but-curious attacker, which wants to learn information about the secret without deviating from the protocol. Modern MPC protocols obtain active security, *i.e.* security against malicious parties which can actively deviate from the protocol. To obtain maliciously secure MPC protocols in the secret sharing model [172], there are several number of approaches. The traditional approach is to use Verifiable Secret Sharing (VSS), which works in the information theoretic model and requires that strictly less than $n/3$ parties can be corrupt. The modern approach, adopted by protocols such as BODZ, SPDZ, Tiny-OT, MASCOT etc. [29, 72, 117, 145], is to function in a full threshold setting (*i.e.* all but one party can be corrupted) and attach

information theoretic MACs to each data item. This approach turns out to be very efficient in the MPC setting, apart from its usage of public-key primitives.

Protection against attacks in the wire-probe model is usually done via masking, which is in many cases the extension of ideas from *passively secure* secret sharing based MPC to the side-channel domain. It is then natural to look at *actively secure* MPC protocols for the extension to fault attacks. We inspired ourselves on the work of SPDZ to design one of our schemes presented in Chapter 5, due to the computational efficiency when using information theoretic MACs and the active adversarial model resistance.

2.6 Boolean Masking

Masking has been the most researched countermeasure against SCA in the literature. Our contribution on SCA countermeasures deals solely with Boolean masking in particular, so, below, we present the state of the art of Boolean masking schemes.

2.6.1 ISW

In their manuscript, Ishai *et al.* [108] proposed a protected AND gate ($c = a \cdot b$), given as

$$z_i = a_i b_i \bigoplus_{j \neq i} (r_{i,j} \oplus a_i b_j) \oplus a_j b_i, \tag{2.1}$$

where $r_{i,j}$ is a uniformly distributed random bit, and $i, j = 1, \ldots, s$.

This scheme is not suitable for hardware implementations since the model does not contemplate non-ideal behaviors from the hardware, like the presence of glitches due to differences in the logic gates propagation times. Hence, it fails to provide security given the unexpected dependencies that appear with glitches.

2.6.2 Threshold Implementations

The first paper to propose a provable secure masking scheme taking into account the effect of glitches was the work of Nikova *et al.* [147]. Known as Threshold Implementations (TI), this masking scheme defines three properties to ensure first-order side-channel security:

- **Correctness**: a shared function $\mathbf{f}$ such that $f_i(\mathbf{x}) = y_i$, where $i = 1, \ldots, s$, is correctly shared if $\sum y_i = y = f(x)$.
- **Non-completeness**: a shared function $\mathbf{f}$ is non-complete if every component function f_i is independent of at least one input share.
- **Uniformity**: The s-sharing of an n-bit permutation $f(x) = y$ is uniform if its sharing $\mathbf{f}(\mathbf{x}) = \mathbf{y}$ is an ns-bit permutation.

Subsequent paper from Bilgin *et al.* [40] extended the first version of TI to higher-order Threshold Implementations (HO-TI). A crucial property is introduced to provide security at higher-orders, namely d^{th}-order Non-Completeness, which is defined as follows:

Definition 2. ***d^{th}-order Non-completeness [40].*** *A shared function* $\mathbf{f}$ *is d^{th}-order non-complete if any combination of up to d component functions f_i is independent of at least one input share.*

When considering higher order protection, it is necessary to ensure univariate and multivariate security. On the one hand, univariate attacks refer to the case where probes are taken in the same clock cycle. On the other hand, multivariate attacks allow the attacker to combine probes across different clock cycles. The scheme from Bilgin *et al.* [40] does not provide the means to achieve higher-order multivariate security. Thus, it is not possible to compose operations secured with this method when considering higher orders of security without additional refreshing.

It is important to note that non-completeness is a necessary condition for security in the presence of glitches at any order of security. On the contrary, uniformity is only necessary for first-order security, and it is neither necessary nor sufficient for higher orders.

In these schemes, the number of input shares for a secure implementation depends on the degree of security d, and the algebraic degree t of the operation to share. The lower bounds of the number of input and output shares such that there always exists a sharing of an algebraic degree t function satisfying the above non-completeness property are given in [40] as follows:

$$s_{in} \geq td + 1, \tag{2.2}$$

$$s_{out} \geq \begin{pmatrix} s_{in} \\ t \end{pmatrix}.$$

It is noteworthy that it is not possible to obtain a first-order secure AND gate using three shares since this structure is not uniform. Additional randomness is needed to obtain uniformity and, thus, security. Since adding fresh randomness is not preferable, an AND/XOR gate (also known as Toffoli gate) is often used, which indeed can be secure with three shares. As an example, we present a first-order secure AND/XOR gate ($t = 2$) $z = a \oplus bc$, with $(d, s_{in}, s_{out}) = (1, 3, 3)$ in Eqns. (2.3).

$$\begin{aligned} z_1 &= a_1 \oplus b_1c_1 \oplus b_1c_2 \oplus b_2c_1, \\ z_2 &= a_2 \oplus b_2c_2 \oplus b_2c_3 \oplus b_3c_2, \\ z_3 &= a_3 \oplus b_3c_3 \oplus b_1c_3 \oplus b_3c_1. \end{aligned} \tag{2.3}$$

Additionally, in Appendix A.2, we present a second-order secure implementation of the same operation, with $(d, s_{in}, s_{out}) = (2, 5, 10)$ in Eqns (A.1).

2.6.3 Consolidated Masking Schemes (CMS)

In HO-TI, $s_{out} > s_{in}$, which means that at some point, there has to be a compression layer, which brings the number of shares (s_{out}) back to the original number (s_{in}). In [157], Reparaz *et al.* demonstrated that it is possible to achieve higher-order multivariate security with TI by inserting fresh randomness to each s_{out} output share right before the compression phase. This supplementary randomness addition, known as *refreshing* or *remasking*, naturally provides uniform sharing.

Additionally, Reparaz *et al.* showed that it is possible to implement a non-linear operation securely on hardware using $s_{in} \geq d + 1$ shares, independent of the degree of the function t. The number of output shares, on the other hand, is lower bounded by a function depending on t, specifically by $s_{out} \geq (d + 1)^t$. As a trade-off, these sharings need an extra condition on the inputs, *i.e.*, they must be *independent* input sharings.

We provide the first-order sharing of an AND/XOR gate ($z = a \oplus bc$) where $(d, s_{in}, s_{out}) = (1, 2, 4)$ in Eqns. (2.4).

$$\begin{aligned} z_1 &= a_1 \oplus b_1c_1, & z_3 &= b_2c_1, \\ z_2 &= b_1c_2, & z_4 &= a_2 \oplus b_2c_2. \end{aligned} \tag{2.4}$$

Similarly, the second-order sharing, where $(d, s_{in}, s_{out}) = (2, 3, 9)$, is provided in Eqns. (A.2).

Note that it is suggested to refresh each output share of the sharing mentioned above to achieve multivariate security. Moreover, unlike TI, the independence of input sharings of each variable (*e.g.* $\mathbf{b}$ and $\mathbf{c}$ in Eqns. (2.4)) is crucial for this scheme to provide security. We say that two inputs are dependent when there exists a relationship between them. For example, in Eqns. 2.4, given a dependent sharing of b and c, such that $\mathbf{b} = \mathbf{c}$, the second term would become: $z_2 = b_1 c_2 = b_1 b_2$. This breaks the non-completeness condition and leaks information of the value b.

2.6.4 Domain Oriented Masking (DOM)

Domain oriented Masking (DOM) was proposed by Gross *et al.* [98]. They propose a similar scheme as CMS, with $s_{in} \geq d + 1$, but optimizing the randomness consumption. A first-order secure AND gate consumes a single fresh random bit given independent inputs.

In the paper, two different multipliers are presented, known as DOM-*indep* and DOM-*dep*. The first one requires independence among the inputs, while the second one does not require this condition. Thus, new registers and randomness are needed in the DOM-*dep* multiplier to break possible dependencies in the inputs.

DOM-*indep*

A general expression for a customary degree of security d is given in Eq. (2.5), for a multiplication ($z = ab$) [98].

$$z_i = a_i b_i + \sum_{j>i}^{d+1} (a_i b_j + r_{i+j*(j-1)/2}) + \sum_{j<i}^{d+1} (a_i b_j + r_{i+j*(j-1)/2}), \tag{2.5}$$

where $i \in [1, d+1]$. Note that only cross-domain terms are XOR-ed or blinded with the random variables.

DOM-*dep*

To solve the problem of the additional condition for the inputs to be independently shared, Gross *et al.* [98] proposed a second secure multiplier,

DOM-*dep*. This multiplier has the advantage of not relying on the independent inputs assumption. Instead of calculating $a \cdot b$ directly, a random blinding value c (shared) is used to calculate it. For correctness, we need a new term for correction, as we can see in Eq. (2.6).

$$a \cdot b = a \cdot \underbrace{(b + c)}_{blind} + \underbrace{(a \cdot c)}_{corr.} \tag{2.6}$$

It is important to note that the recent manuscript form Moos *et al.* [137] presents a flaw in a third-order CMS protected example, also applicable to higher degrees of security. Similarly, they present a flaw in a second-order DOM secured operation, which also applies to higher orders. Note that these findings do not affect the works presented in this thesis, given that we mainly focus on TI, and first and second orders of security. Only the implementation presented in § 3.5.1 might be vulnerable to the flaw presented in [137], but further investigations are needed.

2.7 Logic Simulation and Reliability Testing

Logic and fault simulation play a crucial role in VLSI design, used mainly for functional verification or reliability testing. Outside the cryptography community, the field of integrated circuits testing has developed plenty of simulation tools and analysis techniques over the last decades to examine the effect of possible manufacturing failures in a product, also known as reliability analysis. Testing through simulation has the advantage that it can be performed in great detail and is potentially very accurate since it gives realistic emulation of faults and detailed monitoring of their consequences on the system [176]. Moreover, practical evaluations of a real product need an enormous amount of time as well as considerable expertise. In addition to this, a preliminary evaluation is advantageous in order to reduce the costs of the fabrication process.

2.7.1 Logic Simulation

Logic simulation is a widely researched topic in the literature [21, 49, 59, 103, 126, 127, 134], where logic simulators are classified into two types: compiled or event-driven simulators. On the one hand, the first focuses on optimizing the simulation of individual gates. On the other hand, the second focuses on optimizing the number of gates to evaluate. Compiled simulation evaluates every gate in every simulation, whereas event-driven simulation only simulates

activated gates, implementing a scheduler to keep track of the gates to simulate. There are also several simulation levels depending on the level of modeling used to represent the simulated system, including behavioral level, register-transfer level (RTL), gate level, transistor level, or electronic system-level (ESL).

Event-driven simulation

An event represents a change in the value of a signal, which at the same time activates the component driven by this wire [3]. When a wire is evaluated, all the gates driven by this wire, including fan-outs, are *activated* for subsequent evaluation. The simulation starts evaluating the primary inputs which activate the first gates and propagates the events, storing the active elements in an *activity list* for posterior evaluation, following the order of activation. The propagation finishes when a Primary Output (PO) is reached, *i.e.*, an output of the circuit, or when a register is found. The activity list is a list of pending elements waiting to be evaluated. An active component is evaluated (and then removed from the activity list) only if all its inputs are already evaluated. Otherwise, it is sent back to the end of the queue following the other active elements. The simulation continues until the activity list is empty.

2.7.2 Integrated Circuits Testing

The role of *reliability testing* in digital circuits is to evaluate the correct functionality of a particular design, thereby examining whether something went wrong during the manufacturing process. The role of *diagnosis* is to ascertain the underlying reason for a malfunction [52]. Reliability testing is a crucial part of the design flow of any circuit and is done along different stages of the design to ensure correct functionality. Multiple factors can affect the manufacturing process, leading to the wrong result. Simulation techniques help to anticipate and determine what failed during the process. Fault simulation has been extensively used in fault diagnosis to achieve this goal.

Fault Simulation

In fault simulation, the fault-free circuit is simulated to get the correct output given a specific input test vector. Then, the faulty circuit is simulated with the same input, by reproducing the fault-free circuit with a particular fault injected. If the output of the faulty circuit differs from that of the fault-free circuit, it is said that the given input test vector detects such a fault. Instead, when the effect of the fault does not propagate to the outputs, we conclude

that the fault is not observed and hence, not detected by the given test vector. More formally, for a given test vector, a fault is detectable if it is *controllable* and *observable* [53]. Controllability refers to a set of input vectors that make a difference at the location of the chosen fault between the fault-free and faulty circuits, *i.e.*, the wire where the fault is located has a different value for the fault-free circuit than for the faulty circuit. Observability denotes a set of input vectors that make a circuit's primary output dependent on the value of the location of the chosen fault in the fault-free circuit, *i.e.*, the fault propagates to at least one primary output. The fault is detectable if the intersection of the controllability and observability sets is not empty. Different fault simulation techniques exist in the literature, but the most common ones are [52]:

- **Serial** fault simulation, where every fault is analyzed one at a time.
- **Parallel** fault simulation, which is similar to the previous method, but it takes advantage of the inherently parallel operations of computer words to simulate multiple faults at the same time with bit slicing.
- **Deductive** fault simulation, which propagates the fault lists associated with each wire. A fault list is a set containing the identifier of every fault that produces an error in the given line.
- **Concurrent** fault simulation, where each *good-gate* has "attached" a linked-list of *bad-gates*, or copies of the gate affected by some fault in the inputs or the output, leading to a different function compared to the original gate. The number of faulty gates in the linked-list corresponds to all the different (combinations of) faults in the circuit.

The advantage of the last two techniques is that in one input-output pass, it is possible to asses the effect of every fault, making them highly efficient. However, in the presence of a significant number of faults, the memory demand can be immensely high.

The fault diagnosis of digital circuits is most often addressed with single-fault simulation models because it is infeasible to cover all possible combinations of multiple faults. While many fault simulation tools for reliability analysis of integrated circuits exist, they are commonly limited to one-bit faults. It has been shown that the test vectors set for detecting single (one-bit) faults can also detect a high percentage (higher than 99.6%) of multiple faults [4,52,54,104,109].

2.8 Security Evaluation and Performance Results

To perform the security evaluations of the different designs presented throughout this thesis, we used the leakage assessment methodology known as Test Vector Leakage Assessment (TVLA), and for the deployment, we use the SAKURA-G board [1], which holds two FPGAs.

2.8.1 TVLA

The method introduced in [25], known as non-specific test vector leakage assessment (TVLA), is employed to detect leakage, or similarly, to detect if intermediate variables are correlated with secret data. We decide to use this test since with just a single can catch any source of leakage. This test is not an attack, and retrieving the key is not the aim. This test gives information about the security order and the leakage, whose presence is a necessary condition, but not sufficient for an attack to succeed. The fact that no leakage is observed gives confidence to the designer on the security of the design. Since the laboratory is a completely safe environment, barely affected by noise, we can say that finding no leakage provides a robust security guarantee. However, if leakage is found, this does not warranties that the leakage is exploitable by means of a key retrieval attack.

We mount the experiment known as Fixed-vs.-Random following the methodology described in [94]. We collect power traces from two different groups (one fixed and the other one random) and compare the two sets using the t-test statistic. When the t-statistic surpasses the threshold of ± 4.5, one can conclude with confidence 99.9995% that the two sets of power traces follow a different distribution, and hence, the design leaks. We perform two different tests to see the effect of the countermeasure. We first evaluate with masks off, *i.e.*, with the countermeasure switched off. Then, we switch on the countermeasure and perform the same test with masks on. It is important to note that the fixed plaintext is chosen at random, and we always use that same plaintext for all our experiments. A true leaking point should leak independently of the fixed plaintext chosen, but results might change when changing it. To ensure the true nature of the leakage found, one could repeat the same experiment for multiple fixed plaintexts, averaging the final result. However, this would be highly time consuming, so we perform the experiment only once with a random plaintext.

The basic t-test corresponds to the first-order test, where the first-order moment or the mean of the fixed and the random sets are compared. For the second- and third-order tests, higher order moments are compared, namely, the variance and the skewness respectively.

2.8.2 Setup

To evaluate the security of our implementations, we use a SAKURA-G board designed explicitly for side-channel evaluation. This platform includes two Spartan-6 FPGAs, the control FPGA, and the cryptographic FPGA. To reduce the experimental noise, we split our implementation in the two FPGAs. The control FPGA handles the communication with the host computer and generates the shares for the cryptographic FPGA, and in the crypto FPGA, we deploy the actual cryptographic scheme. This way, we isolate the power consumption of the actual encryption, reducing the noise considerably. We use a very slow 3 MHz clock to ensure clear power traces with minimal overlap between consecutive time samples. The synthesis of the designs is done using the Xilinx 14.7 tools with the `KEEP HIERARCHY` constraint, in order to avoid optimizations across different shares. We measure the power consumption using a Tektronix DPO 7254C oscilloscope, sampling at 1 GS/s.

2.8.3 Results

The performance results gathered throughout this thesis are given using the NanGate 45nm Open Cell Library [144], and synthesized with Synopsys DC Compiler 2017.09. We *compile* with `-exact_map` option, and medium effort for mapping, area, and power to minimize the differences between the written and the synthesized code, and to avoid optimizations that could affect the security.

2.9 Conclusion

Throughout this thesis, we present multiple concepts for the correct design and verification of SCA and FA combined countermeasures, which means that a great variety of concepts for both disciplines, SCA and FA, have to be introduced.

In this chapter, we presented all the concepts that sustain the ideas presented in subsequent chapters. We started with a high-level explanation of the two physical attacks, Side-Channel Analysis (SCA) and Fault Analysis (FA), followed by their respective most relevant attack techniques, known as Differential Power Analysis (DPA) and Differential Fault Analysis (DFA). Then, we described the most relevant countermeasures used to thwart these attacks, hiding and masking for SCA, and Error Detecting Code for FA. Additionally, we provided some information about the state of the art of evaluation tools to asses the security of these countermeasures.

After the higher-level descriptions, we described the concept of actively secure MPC. Then, we presented more in detail several concepts that we will be using in the rest of the manuscript and are essential to understand the new ideas. We further examined a particular family of SCA countermeasures know as Boolean masking, in which we briefly explained the most relevant masking schemes up-to-date. Additionally, we examined the concepts of logic and fault simulation, which are the foundation of our verification tools.

Finally, we have presented the security evaluation methodology that we will be using to test the security of our implementations, together we the laboratory setup.

Chapter 3

Implementations and SCA countermeasures for the Keccak family

In this chapter we introduce key concepts necessary for a successful implementation of SCA protected implementations and the application of these concepts to several algorithms of the KECCAK family. We compare the trade-offs that entail the application of different masking schemes, regarding area, latency, and randomness. Then, we present further concepts on how to correctly apply these techniques for low latency applications.

We start by introducing the KECCAK family algorithms, including the family of permutations KECCAK-f, the Authenticated Encryption scheme KETJE [35], and the standard Secure Hash Function 3 (SHA-3) [2] together with some basis unprotected implementations. Then, we present an FPGA specific unprotected implementation of the SHA-3 algorithm. We continue with the elaboration of two key techniques that help us to achieve SCA secured implementations, namely, the importance of the input dependencies in masked implementations for non-completeness, and the Changing of the Guards [69] technique to achieve uniformity with lower number of shares. Subsequently, we present a family of protected implementations of KETJE algorithms for encryption and decryption procedures. We go further in the analysis of the KECCAK-f permutation and show the importance of independent inputs when securing this algorithm with $d+1$ sharings, and propose a fix to previous state of the art implementations. We then present a novel technique on how to secure unrolled or lower latency

implementations, with some applications to the KECCAK-f permutation.

This chapter is based on respective works published in the conferences Field Programmable Logic (FPL) 2019 [8], Digital System Design (DSD) 2018 [11] and the extended journal version published in Microprocessors and Microsystems (MICPRO) 2019 vol. 71 [13], and Cryptographic Hardware and Embedded Systems (CHES) 2018 [9]. I am the main contributor of all these papers.

3.1 Keccak family algorithms

In this section, we provide an introduction to the KECCAK family algorithms used throughout our work. We present the specifications and unprotected implementations of the underlying permutation KECCAK-f, and the Authentication Encryption scheme and CAESAR candidate KETJE. Finally, we introduce the SHA-3 standard (corresponding unprotected implementations are presented more in detail in the following section).

3.1.1 Keccak permutations

KECCAK is a family of sponge functions using the permutations KECCAK-$f[b]$ where $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ defines different state sizes [32], presented in the NIST SHA-3 standard [163]. Among other properties, KECCAK stands out due to its high performance when implemented in hardware and its great area/speed trade-offs. According to the authors, the security of these variants follows the *Matryoshka principle*, i.e. analysis of KECCAK with small sizes is claimed to be relevant to the variants with bigger sizes.

KECCAK-f operates on a three-dimensional state S, where the bit in the coordinate (x, y, z) is denoted by $S[x, y, z]$. A round consists of five steps:

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta, \text{ with} \tag{3.1}$$

$$\theta: \quad S[x,y,z] \quad \leftarrow S[x,y,z] \oplus \bigoplus_{y'=0}^{4} S\left[x-1, y', z\right] \oplus \bigoplus_{y'=0}^{4} S\left[x+1, y', z-1\right],$$

$$\rho: \quad S[x,y,z] \quad \leftarrow S[x,y,z-(t+1)(t+2)/2],$$

$$\text{with } t \text{ satisfying } 0 \leq t < 24 \text{ and } \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \text{ in } \mathrm{GF}(5)^{2\times 2},$$

$$\text{or } t = -1 \text{ if } x = y = 0,$$

$$\pi: \quad S[x,y] \quad \leftarrow S\left[x', y'\right], \text{ with } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix},$$

$$\chi: \quad S[x] \quad \leftarrow S[x] \oplus (S[x+1] \oplus 1) S[x+2],$$

$$\iota: \quad S \quad \leftarrow S \oplus \mathrm{RC}[i_r], \text{ where } \mathrm{RC}[i_r] \text{ is the round counter in cycle } i_r.$$

Note that we abuse the notation to focus on specific coordinates. Moreover, addition and subtraction in the coordinates are modular. For convenience, we provide visualizations of these round steps in App. A.1 (Fig. A.1) for KECCAK-f[200], which computes 18 rounds. We implement a round-based version of this permutation, which serves as a basis for subsequent implementations. Fig. 3.1 depicts a basic schematic of this architecture.

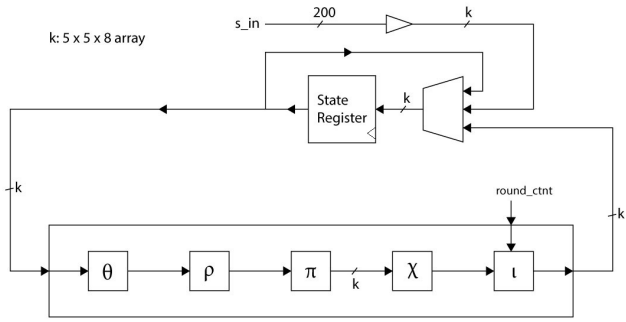


Figure 3.1: KECCAK-p [200] round-based implementation.

Additionally, the authors of [163] specify the KECCAK-p permutations, which are derived from the KECCAK-f permutations and have a tunable number of rounds. They are defined as KECCAK-p $[b, n_r]$, where only the *last* n_r rounds are applied.

3.1.2 Ketje

Ketje [35] is a CAESAR (Competition for Authenticated Encryption: Security, Applicability and Robustness) third round candidate. Although Ketje is not a finalist of the competition, we still believe it is a highly valuable algorithm since it is based on the SHA-3 standard and it is very suitable for side-channel resistant hardware implementations. Ketje is a set of four authentication encryption schemes that allow messages with associated data. They target memory-constrained devices and strongly rely on nonce uniqueness for security. Ketje is built following a layered design, where Keccak-p^* is the underlying permutation, called by the construction MonkeyDuplex, which at the same time is instantiated by the mode MonkeyWrap.

Hereby, we additionally provide the details of the unprotected Ketje implementations. The design process follows a layered structure. We start from the very bottom, with the underlaying permutation Keccak-f (later adapted to Keccak-p^* for the Ketje implementation) which is implemented in a round-based manner, as shown in Fig. 3.1. We continue with the MonkeyDuplex block that calls the permutation. Finally, we implement MonkeyWrap, which instantiates MonkeyDuplex, for encryption, and further extended to support encryption/decryption together. The implementation is made generic, where, by just varying the width of the permutation (b), the four instances of Ketje can be obtained.

Keccak-p^*

Twisted permutations. In Ketje v2 [35], the underlying permutation is modified to the twisted permutation Keccak-$p^* = \pi \circ$ Keccak-$p \circ \pi^{-1}$. The purpose of this updated version of the underlying permutation is to more effectively re-order the state bits. In a round-based implementation, the update from Keccak-p to the twisted permutation can be realized simply by instantiating π and π^{-1} at the end and at the beginning of the round function respectively. Fig. 3.2 illustrates the round function of the twisted permutation.



Figure 3.2: Keccak-p^* round function extending the original Keccak-p

MonkeyDuplex

MonkeyDuplex calls Keccak-$p*$, where different types of calls are supported invoking different numbers of rounds. The performance of the functions can be optimized by reducing the number of rounds, at the expense of requiring nonce uniqueness for the scheme to provide security against key retrieval. MonkeyDuplex follows the structure of a duplex construction, where there is an absorption phase and a squeezing phase alternating. The absorption phase introduces new bits to the state, while the squeezing phase extracts bits from the state. Fig. 3.3 depicts this idea. The function takes as input (absorbs) a binary string of any length σ and returns (squeezes) a binary string of the requested length Z.

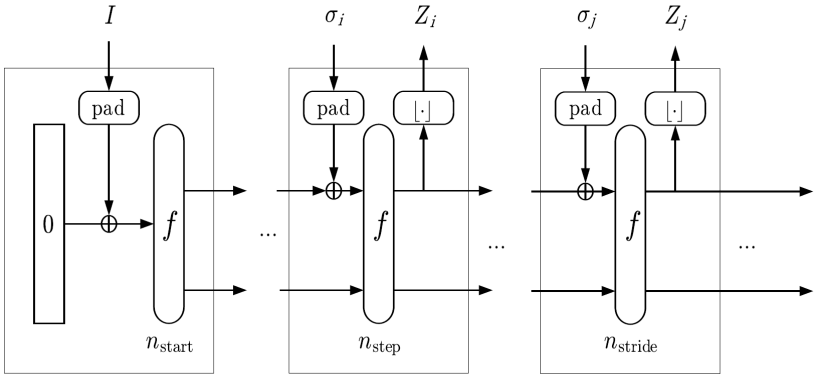


Figure 3.3: The MonkeyDuplex construction [35]

Four parameters (r, n_{start}, n_{step}, n_{stride}) determine the efficiency and security strength of the module. Two types of calls are supported by MonkeyDuplex:

- D.start: at the beginning the string $I = K||N$ that concatenates the Key (K) and the Nonce (N) is introduced in the function after a simple padding is applied, initializing the state. Then, the function f runs for n_{start} rounds, $f[n_{\text{start}}]$.
- D.step or D.stride: the construction absorbs a data block σ , extended with a multi-rate padding until the string length is equal to the *rate* r. Then, either $f[n_{\text{step}}]$ or $f[n_{\text{stride}}]$ is applied to the state f, and afterwards part of this state is extracted (Z).

This construction can be used for mainly three different applications: first, the simplest one, as a stream cipher; second, as a reseedable (fresh seed every block)

pseudorandom bit sequence generator; third, and the most relevant one, as authenticated encryption.

Finally, it is important to stress that both uniqueness and secrecy of I are crucial to comply with the security claims. In addition to this, multiple instances of MonkeyDuplex with the same input I are not allowed, since, otherwise, an attacker could retrieve the full state easily by observing differences in the output.

Implementation. MonkeyDuplex implements the previous three functions in its framework: D.start, D.step and D.stride. All of them call the same permutation, but for different numbers of rounds: $n_{\text{start}} = 12$, $n_{\text{step}} = 1$, and $n_{\text{stride}} = 6$. The first one gets as inputs a string forged with the Key and the Nonce, while the others get a block σ from either the Associated Data or the Plaintext. Only one Keccak-p^* block is instantiated, which means that the input string and the number of rounds are decided based on the function to run. The three aforementioned functions are encoded in a two bit control signal `mode` as: 00, 10 and 11 respectively.

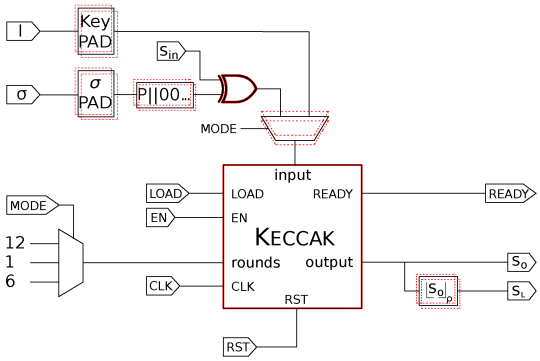


Figure 3.4: Architecture for the MonkeyDuplex implementation. Black datapath for the unprotected version, additional red dotted for the protected.

Two padding blocks are instantiated, one for D.start operation and another one for D.step and D.stride that append a string with the following structure: $10 \ldots 01$ with enough zeros to get the required length. The black datapath in Fig. 3.4 depicts the unprotected diagram of this block.

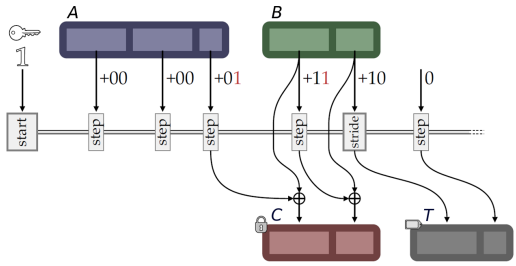


Figure 3.5: The MonkeyWrap construction [35]

MonkeyWrap

The authentication process takes as input a header A or Associated Data, a data body B or Plaintext, and returns a Ciphertext C and a Tag T. The encryption process is known as *wrapping* and the decryption process is known as *unwrapping*. The functionality of the encryption is outlined below and illustrated in Fig. 3.5:

1. The state is initialized with Key and Nonce executing D.start(keypack($K, |K|+16)||N$) where the keypack is a way of encoding the secret key given a string input.
2. The associated data is processed block by block (A_i) by executing D.step($A_i||00$), and D.step($A_i||01$) for the last block of A. The block size ($|A_i|$) of the mode is given by ρ. The last output is XORed with the first plaintext block to get the first ciphertext block.
3. The plaintext is similarly processed with D.step($B_i||11$) and the output of every step is XORed with the next block of the plaintext to get a ciphertext block. The last block of B is treated with D.step($B_i||10$) to produce the first tag block.
4. Subsequent tag blocks are produced by executing D.step(0) until the required T length is achieved.

The decryption is essentially identical to the encryption except for few differences: the scheme receives as input A, C and T, where the same procedure is followed, using the ciphertext data instead of the plaintext. The new tag T' is calculated out of the plaintext blocks decrypted, and finally the entire plaintext B is output only if $T = T'$.

MonkeyWrap supports *sessions*, allowing the encryption of several messages (with the respective associated data) with the same input I. The state is never reset or reinitialized during the same session, so that the function keeps running with the previously updated state. This means that intermediate tags are produced at the end of every encryption, authenticating the current encryption together with the previous ones. Thus, a sequence of messages can be authenticated, rather than a single one. This offers an easy way of sending successive commands while preventing the interference of an attacker.

The Ketje specifications allow four different authenticated encryption schemes, namely: Ketje Jr, Ketje Sr, Ketje Minor and Ketje Major. They are characterized by the underlying permutation and their block size (ρ). They can be used in either lightweight applications or higher-throughput implementations. Tab. 3.1 summarizes the four different possibilities.

Table 3.1: Four Ketje authenticated encryption schemes [35]

Name	f	ρ	Main use
Ketje Jr	Keccak-p* [200]	16	lightweight
Ketje Sr	Keccak-p* [400]	32	lightweight
Ketje Minor	Keccak-p* [800]	128	lightweight
Ketje Major	Keccak-p* [1600]	256	high performance

Important features of the Ketje schemes to take into account:

- As already mentioned above, all but Ketje Major are lightweight.
- The greater block size of Ketje Minor and Ketje Major allows the output of 128-bit tags without extra costs in latency.
- They are very well suitable for side-channel countermeasures and a perfect fit for secure messaging with secured chips, for instance, smart cards.

Implementation. The last block of the building process is MonkeyWrap, which instantiates a single MonkeyDuplex block. The function call can be selected with the `mode` signal, which is decided by a Control module. This module receives two inputs that indicate when the last block of the Associated Data and the last block of the Plaintext are input. Together with a counter, the Control module decides the value of `mode` signal, when to load, what to input, and when to enable MonkeyDuplex. The length of the input σ may vary

depending on the absorbed block. Since in the hardware we can not instantiate a bus to variable number of bits (e.g., first 16 bits and then 20 bits), we declare the bus width driving σ with a fixed maximal length and an integer σ_{ln} is attached to it, indicating the actual length of the bus. Thus, we can correctly apply the corresponding padding. Depending on the encryption stage, when computing the associated data, the plaintext, or the tag, MonkeyDuplex receives as input A_i ($|A_i|$ appended), B_i ($|B_i|$ appended), or 0 (1 appended) respectively.

The output of MonkeyDuplex is then XORed with the next input plaintext block to get the ciphertext, which is output only while the plaintext is being absorbed.

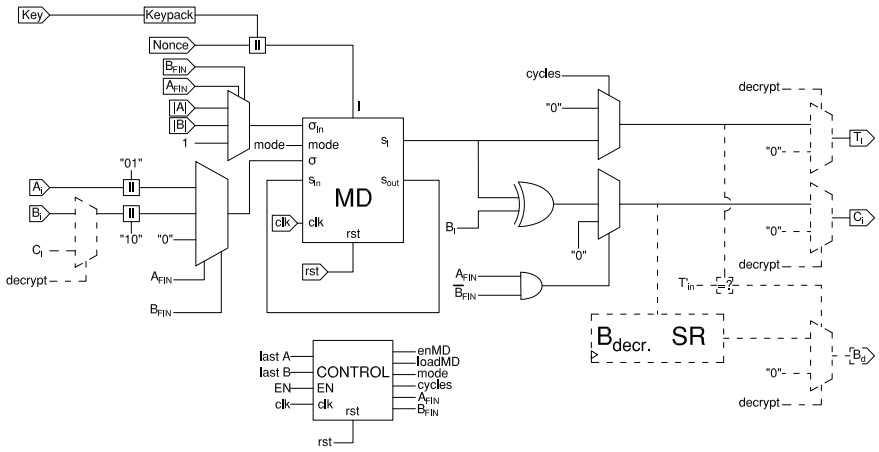


Figure 3.6: The MonkeyWrap implementation including encryption&decryption

Very few extra hardware is needed to extend the encryption block to support *decryption* as well. We add a general control signal `decrypt`, which decides if the block encrypts or decrypts. New control logic is needed to decide the input to MonkeyDuplex, between the input plaintext and the just decrypted ciphertext, after the associated data is fully processed. In order to have the same states as in the encryption procedure, we need to absorb the plaintext, so that is why when decrypting we first get the plaintext block decrypted, and then feed it into MonkeyDuplex to get the next state. According to the specifications, the decrypted plaintext is released all at once, but only after having T' checked against T'_{in}. This is only possible once the new tag T' is calculated entirely, and with the addition of extra registers to save the previously decrypted plaintext.

Thus, optionally add new registers to keep the calculated plaintext until the completion of the tags check. For very long messages, the total size of these registers might become an impractical area overhead, so one can decide to omit them and output the plaintext while being computed. Fig. 3.6 illustrates the schematic of MonkeyWrap, where the unprotected encryption is represented in solid line, and the extension to allow decryption in dashed line.

3.1.3 SHA-3

The Secure Hash Algorithm-3 (SHA-3) was standardized by NIST in 2012, consisting of four cryptographic hash functions and two extendable-output functions. The details of the SHA-3 family are published in FIPS PUB 202 [2].

The SHA-3 algorithm is defined with a sponge construction that operates on top of the Keccak-p permutation. This construction is meant for defining functions on binary data with arbitrary output length and is divided into two phases, absorbing and squeezing. The message is fed into the state block-wise during the absorption, and then the output is *digested* during the squeezing phase. Fig 3.7 illustrates the sponge construction. The length of the absorbed and digested blocks is given by the rate r, while capacity c refers to the size of the remaining state. The padding function appends bits to the input message so that its length is divisible by r.

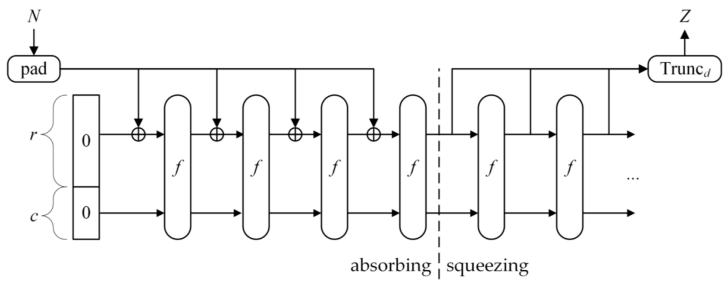


Figure 3.7: Sponge construction. Taken from [2].

Given the three elements f, *pad*, and r, the Sponge$[f, pad, r]$ function on (N, d) is given by Alg. 1, where N is the input string, and d is the output length.

Keccak is the family of sponge contructions utilizing the underlying permutations Keccak-p [b,12+2l], and the padding mechanism "Pad 10*1", where the 0 bit is omitted or repeated enough times so that the padded input message is divisible by r. For a given capacity c, and a given N and d, Keccak

Algorithm 1 SPONGE function

procedure SPONGE[f, pad, r](string N, nonegative int d)
  $P = N||pad(r, len(N))$
  $n = len(P)/r$
  $c = b - r$
  $P_0, \ldots, P_{n-1}$ be the unique sequence of strings of length r such that $P = P_0|| \ldots ||P_{n-1}$
  $S = 0^b$
  For i from 0 to $n - 1$, let $S = f(S \oplus (P_i||0^c))$
  Let Z be the empty string
  $Z = Z||Trunc_r(S)$
  If $d \leq |Z|$ then **return** $Trunc_d(Z)$; else continue
  $S = f(S)$ and continue with Step 8
end procedure

is represented as follows:

$$\text{KECCAK } [c](N, d) = \text{SPONGE}[\text{KECCAK-}p \text{ [1600,24]}, \text{ pad10}^*1, \text{ 1600-}c](N, d)$$

Finally, given a message M, four different SHA-3 hash functions are defined:

$$\begin{aligned}
\text{SHA3-224}(M) &= \text{KECCAK [448](M||01,224)} \\
\text{SHA3-256}(M) &= \text{KECCAK [512](M||01,256)} \\
\text{SHA3-384}(M) &= \text{KECCAK [768](M||01,384)} \\
\text{SHA3-512}(M) &= \text{KECCAK [1024](M||01,512)}
\end{aligned}$$

3.2 Beyond the Limits: SHA-3 in just 49 Slices

In this section, we propose several SHA-3 FPGA specific unprotected implementations, where we explore different trade-offs, significantly improving results from previous works. We introduce a serial FPGA architecture using a row-based processing pattern. Our goal is to push the area constraints to its limits, achieving an extremely compact design occupying just 49 slices, a 44% improvement over the previous smallest design.

On the other side of the spectrum, we present a pipelined implementation, using two internal layers of registers. By placing the registers so that the critical

path is minimized, and by implementing a highly efficient control, we achieve the most area-throughput efficient architecture up-to-date, outperforming the previous best implementation by 70% on a Virtex-6 FPGA.

3.2.1 Related work

A large amount of research has been done on how to efficiently implement SHA-3 in FPGA [106, 110–112, 118, 166, 174, 181]. There are various types of implementations targeting different constraints and trade-offs. From the basic round-based implementation [33], to pipelined and unrolled implementations targeting throughput [106, 174], or serial implementations targeting lightweight applications [111–113, 118, 166, 181], using a folded structure.

Regarding low area designs, different strategies have been suggested. Folded structures are defined based on the orientation and degree of folding, characterized by the folding factor (FF). The very first implementation by Bertoni *et al.* [34] proposed a lane-oriented architecture, where the state is processed lane by lane (see Fig. A.1). The problem of this strategy is that careful control logic is needed to schedule the computations of the θ and χ permutations correctly. Jungk and Apfelbeck [111] proposed the first slice-based folded implementation, where the state is processed slice by slice. This work presents an implementation with FF=8, where eight slices are processed in parallel (state divided into eight parts). A similar approach was used in [112], where different folding factors were explored, with the state implemented using Distributed RAM. Later work of Winderickx *et al.* [181] uses Shift Register Look-up tables (SRL) for the first time, in an implementation using an FF=64 (one slice processed at a time) to achieve a more compact implementation. Finally, the work of Jungk *et al.* [113], presents an even more lightweight SHA-3 architecture also with an FF=64.

For high-throughput implementations, different strategies are used, namely: pipelining, unrolling, or a combination of both. For the former, there are two possible options, *i.e.*, internal or external pipeline: internal pipeline refers to a layer of registers placed inside the round function, in between step mappings, while external pipelining means the registers are placed outside the round function, to keep the state of the permutation or to split unrolled implementations. The work of Ioannou *et al.* [106] presents an unrolled version with two consecutive instantiations of the round function, with an external pipeline in between them. Later work of Sundal and Chaves [174] solely uses an internal pipeline, with no unrolling.

3.2.2 Row-Based Serial Architectures

We illustrate here how we implement the row-based paradigm in our lightweight architectures. Subsequently, we introduce the low-area tailored implementations, where we explore the area-speed trade-off within the row-based paradigm.

Row-based paradigm

In this processing archetype, the computations are performed serially row by row, *i.e.*, one row per clock cycle. Fig. 3.8 illustrates this idea. The row-based paradigm can be seen as a subset of the slice-based paradigm, where calculations are performed on only 5 bits instead of 25. The work of Purnal *et al.* [154] already introduced a row-based implementation, but only for the χ permutation. In this work, we implement the χ and θ permutations in a row-based fashion. We are, to the best of our knowledge, the firsts in the literature to explore this strategy for the θ permutation.

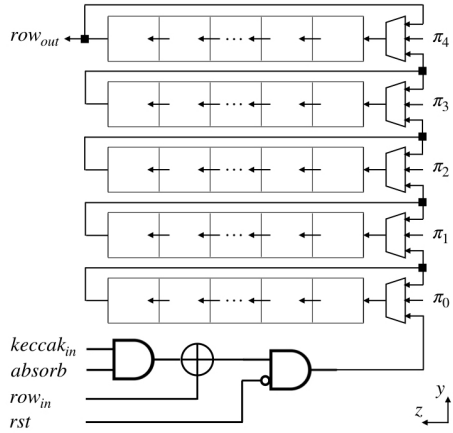


Figure 3.8: Row-based implementation.

The majority of previous works' implementations use Distributed RAM [112, 113, 174] or SRL [181] to store the KECCAK state. We decide to use SRL, which is more area efficient, since we always compute over the last output row of the chained SRLs (cf. Fig. 3.8) as opposed to DRAM, where extra logic is needed to control the memory addressing. The downside of SRLs is that bits are continually changing, entailing a higher power consumption.

Row-based χ. The implementation of χ is straightforward, where the 5-bit Sbox is applied directly to the output row of the SRL, and subsequently fed into the beginning of the shift register.

Row-based θ. The implementation of θ is not as simple as the previous step mapping: there is a need to store the slice parities before subsequent planes are updated with new values, and also to resolve the dependencies across slices. We compute the parities slice by slice together with the θ computation of each row of the first plane. Meanwhile, we store the parity values in five dual-port DRAM cells, which are later accessed in the right order for the computation of subsequent planes. To solve the dependencies across slices for the computation of θ, we use an approach similar to that of [181]. The current slice parity is calculated and stored in a 5-bit register one cycle before the last row is in the output position. In the next cycle, the previous slice parity is calculated, while the current parity is retrieved from the register to evaluate θ in the output row in the current cycle.

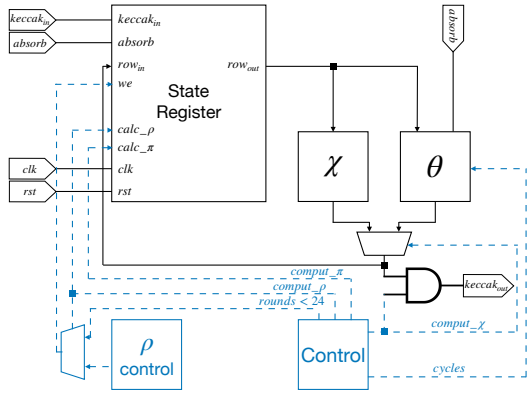


Figure 3.9: SHA-3 lightweight architectures framework.

Slice-based ρ and π. The calculation of the ρ and π permutations is done slice-based, both taking 64 cycles to complete and several MUXs at the beginning of each lane. To calculate ρ, we merely shift the state within each lane for a certain number of cycles, determined by the given offsets. To calculate π, we shift every lane 64 cycles, carefully choosing the feeding lane corresponding to the permutation rearrangement.

To push the area limits even further, we also implement the feeding interface row-wise. The message block is input row by row into the architecture, performing

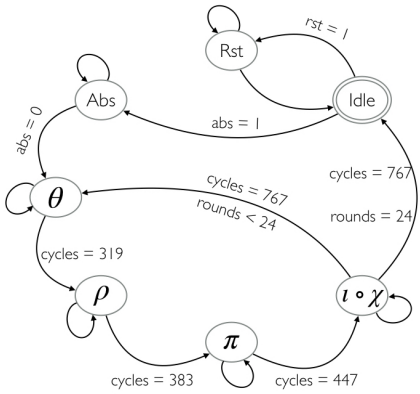


Figure 3.10: SHA-3 lightweight architectures control FSM.

the absorption in the process. This allows us to save XOR operations needed for the absorption and to save additional inputs to the multiplexers already present. The result is an extremely compact architecture. Nevertheless, the trade-off for this area reduction results in a considerable increase in latency. For every operation we perform row-wise, we need 320 clock cycles since the lane size is 64 bits and the state has five planes in total, as shown in Fig. A.1. Thus, the full computation of 24 rounds of Keccak-p takes a total of 18 752 clock cycles: 320 to absorb the input and 768 ($= 2 \times 320 + 2 \times 64$) cycles for each of the 24 rounds.

Architectures

We developed two different lightweight implementations, the first one aiming to balance a good area-frequency trade-off, while the second one focuses solely on optimizing the number of slices.

Both architectures are built following the same structure and differ only in the final optimizations, defined particularly for the desired target. Fig. 3.9 illustrates the basic framework of these implementations. The system is controlled with two input signals, `rst` and `absorb`, and two counters, one counting the `cycles` per round and the other one counting the `rounds`. The Finite-State Machine (FSM) for the control is shown in Fig. 3.10.

Apart from the LUTs used for the SRLs and the DRAMs for θ, the majority of the LUTs belong to control logic. Comparators mostly form this control logic, that decide which permutation should be executed based on the current cycle

and round. The comparators' outputs depend only on the current cycle within a round or the number of rounds already completed. To optimize this logic, we implement those comparators ourselves, trying to squeeze together as many functions as possible into `LUT6_2`s. These LUTs behave as two 5-to-1 look-up tables, allowing the computation of two functions within a single LUT. The condition to successfully instantiate these components is that the two output functions must share common inputs. Provided that all comparators are based on the counter's bits, we were able to highly optimize the different modules. Since the non-optimized design is already quite small, the reduction of every possible LUT has a significant impact on the resources utilized.

SHA3-AF. The SHA-3 Area-Frequency (SHA3-AF) implementation targets a compromise between area and frequency within the row-based paradigm. To achieve this, we explore the use of the components AND2L and OR2L [185], which are latches specially configured to work either as a 2-bit input AND gate with one inverted input or as a 2-bit input OR gate. By using these components, we can reduce even further the number of LUTs used, while not harming the critical path. Combining the use of these components with the instantiation of `LUT6_2`s, we reduce the ρ module from 31 to 17 LUTs, the θ module from 41 to 33, and the control module from 32 to 20 LUTs. Overall, we obtained a final design with 174 LUTs[1]. However, the use of these latches as logic gates comes with several limitations:

- for them to function as logic gates, the input clock signal is set to high, disabling the rest of the flip-flops within the same slice;
- for the same reason, and given that there is a single clock line for the whole slice, SRLs and DRAMS cannot be placed in the same slice together with these components;
- latches from the same slice share the SET and RST signals, so in order to place two or more of these logic gates together, they must have those signals in common.

All these limitations profoundly condition the placement of our components. We implement the latch logic gates, such that they share as many inputs as possible to be able to group them in the same slice and then define multiple location constraints to place everything in the least slices possible. This effort translates into an implementation that occupies 60 slices, 32% smaller than the smallest state-of-the-art implementation [113].

[1] Results for Virtex-6

SHA3-A. The SHA-3 Area (SHA3-A) implementation aims to optimize further the number of slices. Given the aforementioned limitations introduced by the logic latches, multiple slices were instantiated below their full capacity. To solve this, we translate these latches into `LUT6_2` as well, which increases the total number of LUTs and the critical path. Despite the higher number of LUTs (194), now we can distribute them more efficiently with the right placement constraints, achieving our smallest design, mapped into 49 slices. To the best of our knowledge, this is the smallest SHA-3 FPGA implementation to date, with a 44% reduction in area compared to [113].

3.2.3 Area-Throughput Efficiency

In this subsection, the efficiency-tailored implementation is presented, which aims to optimize the throughput-area factor.

In order to evaluate an implementation performance taking into account both area and throughput, the throughput per area efficiency parameter is used, calculated as:

$$E = \frac{T}{A} = \frac{r \cdot f}{L \cdot A}, \tag{3.2}$$

where r is the rate, f is the frequency in MHz, L is the latency in clock cycles, and A is the area in number of slices.

With this architecture, we aim to minimize as much as possible this parameter. Looking at Eq. (3.2), our goal is to maximize the frequency (f) without affecting the latency (L) and the area (A). Unrolling strategies reduce the latency proportionally to the unrolling factor; however, the area is increased similarly, and the maximum frequency severely affected. The decrease in frequency can be mitigated with external pipelines, but this does not result in an improvement concerning the initial frequency. The unrolling plus external pipeline strategy results in a significant throughput increase, but not in efficient area-throughput solutions. Hence, we choose an internal pipelined implementation, with which we double the frequency while barely increasing the number of slices since we only increase the number of registers. The design process consists of optimizing the critical path, by placing the pipelined registers in the right point, as well as deciding how many pipelines are optimal to achieve the best efficiency possible.

In a round-based implementation, it is trivial that the critical path is determined by the permutation layers, particularly by θ and χ. We disregard the use of more than two layers of registers since that profoundly prevents the synthesizer from optimizing logic together, resulting in greater area and longer critical path

in comparison. We explore various architectures using two pipelined stages, and we examine the maximum critical paths of several possible locations of the register layers. As a result, isolating the θ permutation from both sides resulted in the optimal choice, with higher frequency and a smaller area than other options. Fig. 3.11 illustrates this idea and presents the structure of our SHA-3 Throughput-Area (SHA3-TPA) implementation. Sundal *et al.* [174] arrived to similar conclusions, *i.e.*, to use two layers of registers and place the second layer at the end of θ (see Fig. 3.11). However, they do not provide any details regarding the location of the first layer of registers.

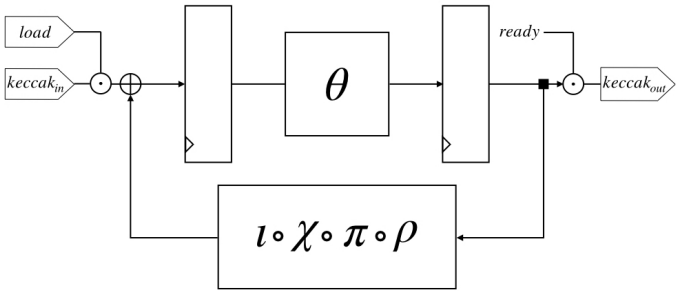


Figure 3.11: SHA3-TPA architecture framework.

The computation of θ determines the critical path in the SHA3-TPA implementation. It is interesting to note that the frequency we obtain in this design is much higher than that in the lightweight designs presented in the previous section, which might be counterintuitive at first sight, since these architectures are around 30x times smaller. In FPGA, the critical path is defined by the number of subsequent LUTs present on it. Despite the high amount of logic used in the pipelined implementation, SHA-3 is known to be highly efficient in hardware, where most of the bits are computed in parallel so that the logic depth and the critical path are shallow. This makes the control an important part of the design to take into account in order to keep the critical path low. The reason for the lightweight implementations to be slower is the need for intricate control logic, which highly affects the critical path.

For SHA3-TPA, we implement a very simple controller, with a counter that is updated based on a t-flip-flop. The counter is updated every time the t-flop has value '1', and the two stages are computed alternatively in respective states of the t-flop. Although the latency of the design increases to 48 clock cycles versus the regular version, the throughput is barely affected, since two messages can be computed at once. We need an extra clock cycle to output the block from the second message, which means 49 cycles in total. Note that the input is always XORed with the output of the permutation. When absorbing the

very first block of the messages, we need a small tweak in the inputs, since the state includes the first round constant (0x01), XORed in ι. To cope with this, we XOR 0x01 to the leftmost byte of the first input block. This tweak is not necessary for subsequent blocks since they will be XORed directly with the output result of the permutation.

We implement this design in both Virtex-5 and -6, to make a fair comparison with previous works since frequency changes considerably between platforms. Additional results for Virtex-7 are also provided. Our SHA3-TPA implementation achieves an efficiency of 16.24 M/s (Mbps/slice) in the first platform, and 28.6 M/s in the second one, which translates into the most efficient SHA-3 implementations tailored to FPGA in both platforms published to date.

3.2.4 Results

Table 3.2: Post-Place and Route Results for SHA3-256. Results given for this and previous works for Virtex-5, -6 and -7, and Spartan-6.

Design	FPGA	Area(GE)			Latency	Max.Freq.	Throughput	Tp/A
		LUTs	Registers	Slices	(cycles)	(MHz)	(Gbps)	(M/s**)
Folded Architectures								
SHA3-A	V-6	193	41	**49**	18 752	197.85	0.011	0.23
SHA3-AF	V-6	174	71	**60**	18 752	**424**	0.025	0.41
New/Bit [113]	V-5	345	84	88	2689	252	0.101	1.14
New/Slice [113]	V-5	364	95	90	1665	257	0.167	1.85
New/Slice [113]	V-6	359	107	91	1665	311	0.203	2.23
Winderickx *et al.* [181]	V-5	-	-	134	-	248	0.156	1.16
Folded [174]	V-5	-	-	476	96	200	2.27	4,76
Pipelined Architectures								
SHA3-TPA	**V-5**	3 344	5 887	1 507	49	551.15	24.475	**16.24**
Pipelined [174]	V-5	-	-	1 163	48	273	14.73*	11.45*
SHA-3 2-stage [106]	V-5	-	-	(2 652)	24	352	31.9*	(12.03*)
SHA3-TPA	**V-6**	4 635	5 887	1 171	49	754.32	33.5	**28.6**
SHA-3 1-stage [106]	V-6	-	-	(1 115)	24	412	18.67*	(16.75*)
Additional results†								
SHA3-A	S-6	200	45	50	18 752	82.15	0.005	0.095
SHA3-AF	S-6	178	77	60	18 752	157.85	0.009	0.15
SHA3-TPA	S-6	5 238	5 908	1 311	49	302.024	13.41	10.23
SHA3-A	V-7	200	40	50	18 752	221.96	0.013	0.26
SHA3-AF	V-7	183	77	60	18 752	417.63	0.024	0.4
SHA3-TPA	V-7	5 258	5 895	1 673	49	832.848	36.985	22.11

*Results scaled to match SHA3-256
()Designs not including Absorption
**M/s = Mbps/Slice
† Results not included in Fig. 3.12

Hereunder, we summarize the implementation results for all the architectures and provide a more detailed comparison with previous works. The results are given for post Place and Route synthesis in Xilinx 14.7, with various

optimizations settings, using different constraints to attain the most optimal designs.

Related works use Virtex-5 (V-5) or Virtex-6 (V-6) for deploying their designs principally. To be able to compare with them, we continue using these same platforms, particularly xc5vlx50t and xc6vlx75t respectively. Since the 6-series FPGA families are newer technology, the majority of our designs are implemented on Virtex-6. Moreover, the 7-series FPGA families use the Virtex-6 technology [184], and thus, our designs are easily portable to any of these FPGA platforms. In order to enable comparison with future works and to avoid outdated results, we additionally provide the performance results of our experiments deployed on Virtex-7 (V-7).

Tab. 3.3 presents the results for our designs and several previous works of interest. All the results in the table are obtained for SHA3-256, with a rate of $r = 1088$, and for a single block of the message. Previous works implementing different algorithms are scaled accordingly by multiplying with a factor $s = 1088/576$, where 576 is the block size (or rate) of SHA3-512. Fig. 3.12 presents a graphical representation of the synthesized results.

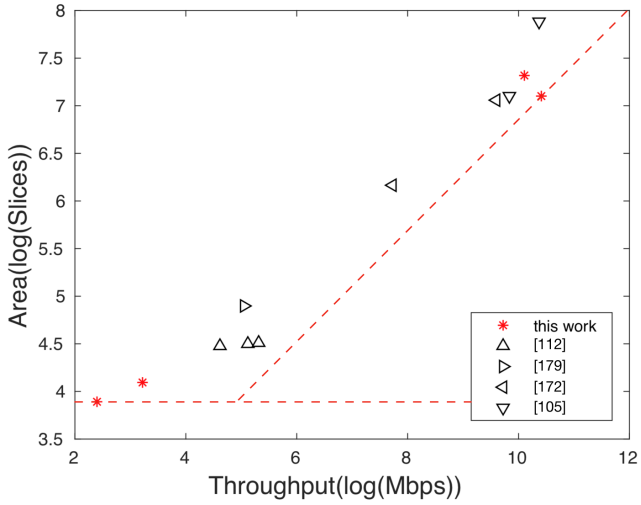


Figure 3.12: SHA-3 synthesis results.

Low Area. Since most of the elements we used in V-6 are also present on V-5 platforms, we also compare our area-constrained architectures with other works that use V-5. The only elements not portable to V-5 are the gate latches AND2L and OR2L. Although not available as a primitive, there are still similar

latches in V-5, *i.e.*, LDCPE, that can be instantiated in the same way to have identical functionality.

As we can see from the table, our SHA3-AF implementation occupies 174 LUTs and 60 slices, a reduction of 32% in the number of slices from the best previous work [113] on V-5, and 34% on V-6. With a maximum frequency of 424 MHz, we achieve an excellent compromise between area and speed.

On the other hand, our SHA3-A implementation takes 194 LUTs and just 49 slices of resources, being the smallest FPGA tailored implementation to date to the best of our knowledge. SHA3-A presents an area reduction over the best previous work of 44.3% in the number of slices on V-5, and 46.2% on V-6.

However, these area improvements naturally translate into a high increase in the latency, resulting in 18 752-cycle implementations, and reducing the area-throughput efficiency. Area-reduced implementations cannot compete with unrolled or pipelined implementations in terms of efficiency. The increase in clock cycles and the decrease in frequency is more dramatic than the decrease in area, the first two affecting the efficiency parameter greatly. Nevertheless, we believe the area reduced architectures would have comparably better area-throughput efficiency results on masked implementations, since then the area entails a highly more significant impact, greatly penalizing lower latency implementations.

Optimized efficiency. For the area-throughput efficient architectures, we compare V-5 and V-6 implementations separately to make a fair comparison, since frequency changes substantially from one platform to the other.

Our SHA3-TPA implementation achieves an efficiency of 16.24 M/s in V-5, a 41.8% improvement over [174], and 35% over [106]. In V-6, SHA3-TPA has an efficiency of 28.6 M/s, a 70.75% improvement over [106], which, we recall, did not include the absorption in its architectures. This implementation results in the most area-throughput efficient FPGA implementation of SHA-3 published to date.

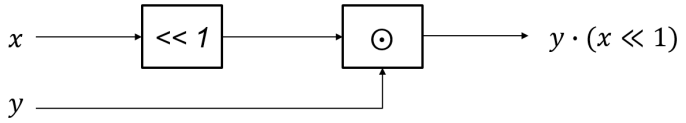
3.3 Towards Secure Hardware

In this next section, we provide an overview of the concepts needed and used throughout this chapter to secure a hardware design. First, we elaborate on the input dependencies, where the importance of this concept in relation to the non-completeness property is addressed, and secondly, on the technique known as “Changing of the Guards” to complement sharing schemes for achieving uniformity using less number of shares.

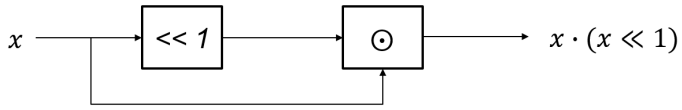
3.3.1 Input dependencies

As already introduced in previous chapter, CMS [157] and DOM [98] are two hardware masking schemes derived from TI. They both aim to reduce the number of shares while maintaining the same order of security. To achieve this, apart from the need of extra fresh randomness, a new condition must be satisfied, i.e., the inputs have to be independent of each other, as specified in § 2.6.3. To satisfy this condition in every operation is not trivial, since even linear operations can hinder it. Failing to provide independent inputs highly compromises the security of the design.

To illustrate the importance of these input dependencies, we present a very simple example where security is broken due to dependent inputs when a CMS/DOM-like sharing scheme is used. For the sake of clarity, we also provide the secure sharing in which the inputs are independent. The example uses two shares to secure against first-order SCA. Fig. 3.13a depicts the example circuit $z = y \odot f(x)$, and Fig. 3.13b the circuit $z = x \odot f(x)$. We choose f to be a simple left shift, $f(x) = x \ll 1$, but any linear function would serve.



(a) Independent inputs operation



(b) Dependent inputs operation

Figure 3.13: Independent and dependent inputs CMS/DOM sharing example

Eqs. (3.3) and (3.4) present in detail the shared operations for the examples in Fig. 3.13 respectively.

$$
\begin{aligned}
\vec{y} &= (y_1, y_2) \\
\vec{x} \ll 1 &= (x_1 \ll 1, x_2 \ll 1) \\
\vec{y} \odot (\vec{x} \ll 1) &= \begin{cases} y_1 \cdot (x_1 \ll 1) \\ y_1 \cdot (x_2 \ll 1) \\ y_2 \cdot (x_1 \ll 1) \\ y_2 \cdot (x_2 \ll 1) \end{cases}
\end{aligned} \tag{3.3}
$$

$$
\begin{aligned}
\vec{x} &= (x_1, x_2) \\
\vec{x} \ll 1 &= (x_1 \ll 1, x_2 \ll 1) \\
\vec{x} \odot (\vec{x} \ll 1) &= \begin{cases} x_1 \cdot (x_1 \ll 1) \\ x_1 \cdot (x_2 \ll 1) \\ x_2 \cdot (x_1 \ll 1) \\ x_2 \cdot (x_2 \ll 1) \end{cases}
\end{aligned} \tag{3.4}
$$

Since the multiplication is a non-linear operation, in the computation there appear four cross terms that would later be compressed back to two shares. We stop our computations at this stage, as we observe that in Eq. (3.4) there is a non-completeness violation in two of the cross terms, where all shares of x are combined together. This failure appears given that the inputs fed into the multiplier are not independent. In Eq. (3.3), where the inputs are independent, even though two different shares are used in an operation, these shares are not of the same variable so no secret data can be reconstructed.

TI prevents this from happening, given that there is always at least one share missing in every share of the function (see Eq. (2.3)). This motivates the assumption that only by sequentially adding linear operations into TI secure implementation would remain secure and no leakage would be introduced.

This concept is of particular importance in KECCAK, given the intricate shape of the initial linear permutation θ. We elaborate on this in subsequent Sect. 3.5.1, analysis that lead us to find a problem in a DOM-KECCAK implementation from [100] due to dependent inputs.

3.3.2 Changing of the Guards

Seminal work in TI [147] proposed a general first-order SCA secure scheme applicable to any operation using three shares. However, a normal AND gate can not be protected simply by using three shares due to the impossibility of achieving uniformity. Hence the Toffoli (AND plus XOR) construction was widely used, where the extra XORed variable could provide uniformity in the three-share construction. The first side-channel protected implementation of KECCAK-f S-Box was proposed by the original authors in [31], which was proved insecure due to the lack of uniformity. In the work of Bilgin *et al.* [38], different sharings were proposed for the KECCAK-f S-Box, including a four-shares uniform scheme.

In [69] Daemen proposes a method to achieve uniformity in a three-shares KECCAK S-Box by just appending few dummy bits to the states, also known as "guards". This method uses these first bits to randomize the first S-Box x^1, then a few of the output bits of this S-Box are similarly used to randomize the second S-Box x^2, and so on until the last S-Box x^j. The last output bits will be the new "guards" of the next round. By feedforwarding these bits and the small extension of the shares, uniformity in the whole S-Box layer is achieved instead of making every single shared S-Box uniform. Fig. 3.14 illustrates a simple example of a sequence of three shared S-Boxes with three shares each.

This technique has been used also to protect ASCON and KEYAC [165], and also in a first-order secure AES [180].

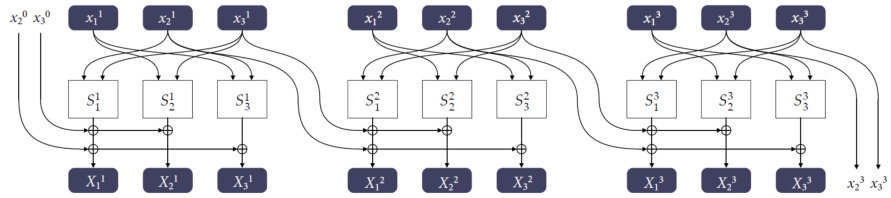


Figure 3.14: "Changing of the Guards" structure for three adjacent S-Boxes from a three shares secure implementation [69]

3.4 Guards in Action: First-Order SCA Secure Implementations of Ketje

In this section, we present a new protected Keccak-f implementation with new trade-offs: slightly bigger area compared to the smallest state-of-the-art parallel implementations [99], but two and three times lower latency. Additionally, we introduce a protected implementation of the whole Ketje family. Furthermore, we extend these implementations to allow encryption and decryption in the same module. We deploy the first-order protected Keccak-f in FPGA and test it, proving the implementation secure for up to 100 million traces. We expand this implementation to get the secure Ketje implementation that inherits the previously proven security, since the expansion only involves linear operations and we use a TI sharing scheme. Moreover, we perform the empirical evaluation on the protected Ketje encryption implementation, demonstrating that indeed this implementation inherits the underlying scheme security. Finally, we find that Ketje provides certain SCA security, up until a limited number of computations reusing the same key.

3.4.1 Secured Implementations

Hereby, we first present a protected round-based Keccak implementation, using the Changing of the Guards technique previously presented, in addition to three shares TI masking. Secondly, we extend the secure Keccak architecture to devise a secure Ketje implementation.

Protected Keccak-*f*

There are several protected implementations of Keccak-f proposed in the literature [31, 38, 99]. The first two use TI to secure their implementations: the first one was shown to use a non-uniform sharing, which was fixed by Bilgin *et al.* in the second one by using a four shares scheme instead of three to achieve uniformity. Gross *et al.*, in [99], propose several serial and parallel implementations using DOM to secure their designs. Ketje is designed to operate best with a parallel implementation of Keccak-f to get the best trade-off between throughput and area. Ketje makes one call to the D.step function for every block in the Associated Data and in the Plaintext. To avoid excessive increase in latency, we seek to compute D.step within one clock cycle. We discard implementations using DOM since a single round of Keccak-f would need two or three cycles to be computed in the parallel implementations. We then go with TI to get the best latency, and we apply the "Changing of the

Guards" method explained before, to get a uniform sharing with three shares. With this method we are able to optimize area with respect to the four-shares version, and no extra randomness is needed. Thus, no dedicated PRNG is needed to feed online randomness.

Securing linear operations in TI is trivial, where just mere repetition is needed to get correctness. Hence, we focus on how to secure the non-linear operation χ and how the guards are implemented. Eq. (3.5) shows the equations followed in our first-order implementation with three shares, similarly given in [69].

$$
\begin{aligned}
& \text{For } 0 \leq b \leq 2\text{:} \\
X_1^j[b] &\leftarrow x_3^j[b] \oplus (x_3^j[b+1] \oplus 1)x_3^j[b+2] \oplus x_3^j[b+1]x_1^j[b+2] \oplus x_1^j[b+1]x_3^j[b+2] \ for\ j > 0 \\
X_2^j[b] &\leftarrow x_1^j[b] \oplus (x_1^j[b+1] \oplus 1)x_1^j[b+2] \oplus x_1^j[b+1]x_2^j[b+2] \oplus x_2^j[b+1]x_1^j[b+2] \ for\ j > 0 \\
X_3^j[b] &\leftarrow x_2^j[b] \oplus (x_2^j[b+1] \oplus 1)x_2^j[b+2] \oplus x_2^j[b+1]x_3^j[b+2] \oplus x_3^j[b+1]x_2^j[b+2] \ for\ j > 0 \\
& \text{For } b = 3, 4\text{:} \\
X_1^j[b] &\leftarrow x_3^j[b] \oplus (x_3^j[b+1] \oplus 1)x_3^j[b+2] \oplus x_3^j[b+1]x_1^j[b+2] \oplus x_1^j[b+1]x_3^j[b+2] \oplus \\
&\quad x_3^{j-1}[b] \ for\ j > 0 \\
X_2^j[b] &\leftarrow x_1^j[b] \oplus (x_1^j[b+1] \oplus 1)x_1^j[b+2] \oplus x_1^j[b+1]x_2^j[b+2] \oplus x_2^j[b+1]x_1^j[b+2] \oplus \\
&\quad x_2^{j-1}[b] \ for\ j > 0 \\
X_3^j[b] &\leftarrow x_2^j[b] \oplus (x_2^j[b+1] \oplus 1)x_2^j[b+2] \oplus x_2^j[b+1]x_3^j[b+2] \oplus x_3^j[b+1]x_2^j[b+2] \oplus \\
&\quad x_2^{j-1}[b] \oplus x_3^{j-1}[b] \ for\ j > 0 \\
X_2^0[b] &= CG_2[b-3] \\
X_3^0[b] &= CG_3[b-3]
\end{aligned}
\tag{3.5}
$$

The S-Box index j assumes a one-dimensional array extracted from the three-dimensional state of Keccak (Fig. A.1). For each plane in the abscissa axis (x) of the state, an array is derived by unrolling the plane, where $0 < j \leq y + 5z$.

It is difficult to compare our implementations area-wise with previous works given that different synthesis libraries were used. However, latency can be compared precisely. Since no registers are needed to secure the non-linear operation, the corresponding KECCAK-f implementation takes respectively two times and three times less cycles than the PARALLEL double clocked and the PARALLEL pipelined implementations from [99]. On the other hand, our implementation is between 6% to 28% (depending on the library) larger in area than the previous works. The protected implementation of the twisted permutation KECCAK-p^* is trivially derived from the secured KECCAK-f: additional blocks π and π^{-1}, for which plain repetition suffices to secure them since both are linear operations, and small change in the control logic to decide the number of rounds from the input are needed.

Securing Ketje implementations

To protect the last two layers of KETJE is easier than the underlying permutation since only linear operations are used. Fig. 3.4 represents the protected implementation of MONKEYDUPLEX, where the red dotted datapath represents the (additional) components needed to secure the implementation. We instantiate the secure KECCAK-p^*, and to secure the linear modules, simple repetition suffices. Note that for the padding we also instantiate it three times; however, this would not work in the case of a design with even shares because the padded string would be canceled when unsharing. In general, it is enough (mandatory in the case of even number of shares) to apply the padding to just one share and pad the rest of the shares with all zeros. The same procedure applies when securing an addition with a constant.

We follow the same procedure to secure MONKEYWRAP: the protected MONKEYDUPLEX is instantiated, and the surrounding logic including Muxes, paddings, the shift register, and the comparator are repeated three times, one per each share.

The control signals do not carry sensitive information, hence we do not need to protect them nor protect the Control block. When securing the decryption datapath, a problem arises. How do we compare the new calculated tag shared with the input tag? To do the comparison, in the case it is implemented, we have to unshare the calculated T' as both tags are most certainly shared with different random masks. Reconstructing the tag from the shares would result in a *non-completeness* violation, where a single wire (probe) would carry information of all shares. This does not entail a problem since the attacker already knows the tag, which is given as input to the decryption. Probing,

and getting a different tag, just means that the attacker learns something went wrong in the authentication process, but he never learns secret data.

3.4.2 Evaluation

To test our implementations in practice, we deploy them into FPGA and conduct the leakage detection experiment previously introduced in § 2.8. No information leakage is detected with up to 100 million traces.

Keccak-*f* evaluation

From all the designs, we deploy the smallest KECCAK-f to test, where $b = 200$. Given the properties of the permutation, the conclusions on a specific version of KECCAK-f transfer to versions of different width, according to the authors of [32]. This property is known as the Matryoshka principle. Since KECCAK-f does not have a key or plaintext as inputs like a regular cipher, we treat the whole input state as secret data. In the evaluation we perform, this translates in the first group being computed with a fixed input, while the second one is computed with a random input that changes for every computation.

We first test the unprotected implementation for 20k traces to verify that our setup is sound and we can find leakage, which is expected in every order. We then activate the countermeasure and run the test again. Now only second-order leakage is expected, since our design has first-order protection. Fig. 3.15 illustrates this evaluation results. Threshold is represented by the red lines.

Ketje evaluation

To complete the full KETJE implementation, only linear operations are used on top of KECCAK-p^*. As shown in Sec. 3.3.1, linear operations prior to non-linear operations might introduce dependencies among shares and hence non-completeness failures. Based on the conclusions of that section, we can assume that the KETJE implementation is secure as well, since we use TI to protect it. In the following, we present the practical experiments for the KETJE encryption implementation, confirming our assumptions, and showing that our implementation is secure with up to 100 million traces.

KETJE does have key and plaintext as inputs, so this time we consider solely the key as secret input. We similarly split the computation over two groups, but here both groups receive the same fixed key. Then, both plaintext and

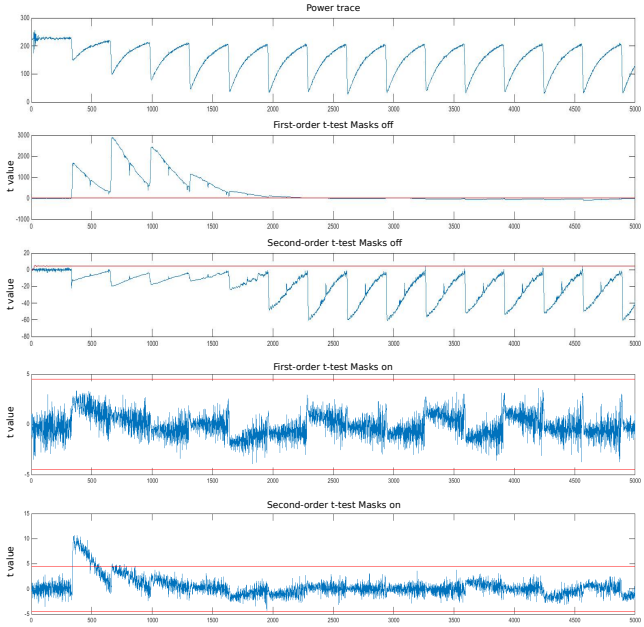


Figure 3.15: Welch t-test evaluation of full KECCAK-f. Top to bottom: power trace, first- and second-order masks off test, and first- and second-order masks on test

associated data are fixed for the first group, while for the second group, they are all random and different in every computation. Thus, we want to see that for the same secret data (key) the computation over the fixed group behaves as random, without revealing any information on the key. Fig. 3.16 presents the practical evaluation for the complete implementation. Following the same principle as before, we first switch off the countermeasure ("masks off"), to see that the implementation leaks already from first-order as expected. Finally, we switch on the countermeasure and confirm that there is no leakage to be seen in any order.

There are several interesting observations to note in this experiments. The first observation is that the leakage in the "masks off" experiment appears much later than usual. The plot of this experiment is for 1 million traces, compared to 20k traces in the KECCAK experiment. Thus, we can empirically confirm that KETJE provides limited SCA resistance, as stated in [35], but only up to certain number of computations where the same key is reused. In real applications, the same key should never be used for a significant number of computations. In

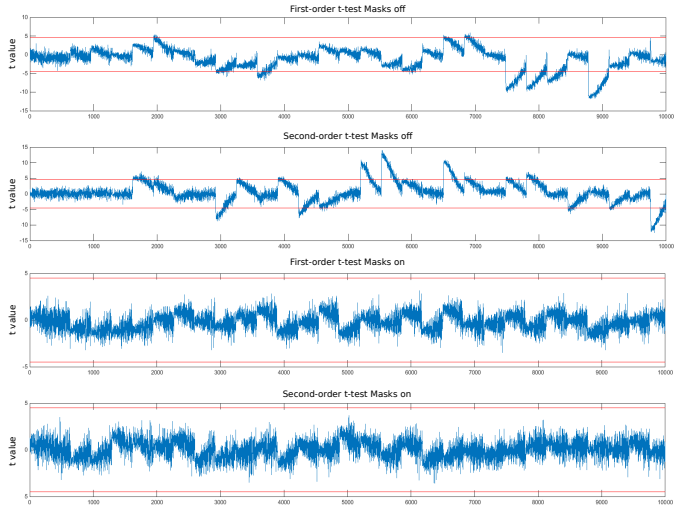


Figure 3.16: Welch t-test evaluation of full KETJE encryption. Top to bottom: first- and second-order masks off test, and first- and second-order masks on test

our experiments we eventually find leakage, since the key is never changed.

The next important observation is that no second-order leakage appears despite our implementation is only first-order secure. Given the small resistance that we observe in the KETJE encryption itself, by introducing the actual side-channel protection on top, we get that leakage does not even appear in the second-order evaluation. Following the same reasoning as before, we presume that leakage will be detected in the second order with enough traces.

3.4.3 Results

Here we present the synthesis results for all the implementations: area, latency, max. frequency, and power. Other than the four bits appended to the state to act as "guards", which are given together with the initial sharing, there is no extra randomness needed. Tab. 3.3 summarizes them.

Three shares are used to secure our implementations. Intuitively, the protected implementations should be three times larger in area than the unprotected ones. Instead, they are around 3.6 times larger. The extra overhead comes from the more complex implementation of the non-linear function.

Table 3.3: Synthesis results for the entire KETJE family using the NanGate 45nm Open Cell Library

Design	State	Area(GE)		Latency	Max.Freq.	Power
	(bits)	χ	Total	(per round/block)	(MHz)	(mW)
Protected KECCAK-p^*						
p^*[200]	200	6 319	15 113	1	892.85	1.58
p^*[400]	400	12 639	30 104	1	892.85	2.87
p^*[800]	800	25 279	60 092	1	909.1	5.47
p^*[1600]	1 600	50 346	119 816	1	877.2	9.96
Unprotected KETJE encryption						
JR	200	707	5 447	1+1+6+8	632.9	0.682
SR	400	1 413	9 663	1+1+6+4	591.7	1.12
MINOR	800	2 827	19 665	1+1+6+1	585	2.34
MAJOR	1 600	5 653	37 650	1+1+6+1	543.5	4.13
Protected KETJE encryption						
JR	200	6 319	18 335	1+1+6+8	892.85	2.08
SR	400	12 639	35 136	1+1+6+4	892.85	3.63
MINOR	800	25 279	73 516	1+1+6+1	909.1	7.75
MAJOR	1 600	50 346	144 022	1+1+6+1	877.2	14
Unprotected KETJE encryption&decryption						
JR	200	707	6 109 ($+3.6 \cdot \#B + 5.67 \cdot \#B$)	1+1+6+8	632.9	0.93
SR	400	1 413	10 276 ($+5.6 \cdot \#B + 5.67 \cdot \#B$)	1+1+6+4	591.7	1.29
MINOR	800	2,827	20 554 ($+19.6 \cdot \#B + 5.67 \cdot \#B$)	1+1+6+1	585	2.7
MAJOR	1 600	5 653	39 596 ($+40 \cdot \#B + 5.67 \cdot \#B$)	1+1+6+1	543.5	4.86
Protected KETJE encryption&decryption						
JR	200	6 319	20 032 ($+3 \cdot 3.6 \cdot \#B + 3 \cdot 5.67 \cdot \#B$)	1+1+6+8	892.85	2.8
SR	400	12 639	36 645 ($+3 \cdot 5.63 \cdot \#B + 3 \cdot 5.67 \cdot \#B$)	1+1+6+4	892.85	4,24
MINOR	800	25 279	77 608 ($+3 \cdot 19.6 \cdot \#B + 3 \cdot 5.67 \cdot \#B$)	1+1+6+1	909.1	9.33
MAJOR	1 600	50 346	145 259 ($+3 \cdot 40 \cdot \#B + 3 \cdot 5.67 \cdot \#B$)	1+1+6+1	877.2	16.4
Related work [99]						
Protected KECCAK						
PARALLEL pip.*	1 600	57 600	111 800	3	837.5	-
PARALLEL d.c.*	1 600	44 200	100 500	2	803.9	-
PARALLEL pip.**	1 600	50 400	97 700	3	846.7	-
PARALLEL d.c.**	1 600	38 400	85 700	2	877.2	-

*Synthesis library UMC 130nm
**Synthesis library UMC 90nm

For the encryption&decryption implementation results, we report the version where the decrypted plaintext is released on the fly as the main architecture. Note that the area increase of the encryption&decryption scheme is minimal compared to the just-encryption module. Additionally, as an optional choice and to comply with the specifications, we provide the results for the implementation where the plaintext is not revealed until the tags are verified (included in parenthesis in Tab. 3.3). We implement a shift register that stores the decrypted plaintext block by block. To be able to decrypt any message in the session, the shift register has to be the size of the longest ciphertext message. This means the area of the decryption depends on the length of the ciphertext/plaintext. The area of the shift register is given in function of the number of plaintext

bits $\#B$, where the expression has two coefficients: the first one refers to the combinational area per bit, calculated experimentally, and the second one refers to the sequential area, 5.67 GE being the area of a single bit register.

We use a special notation to indicate the latency in the KETJE implementations: Cycles per block for processing the Associated Data + Cycles per block for the Plaintext + 6 (stride cycles) + #cycles to produce 128 bit tag. We do not include here the cycles needed to initialize a session, which are always 12 cycles. In encryption mode the Ciphertext is available after Associated Data and Plaintext have been processed. In decryption mode both Plaintext and Tag are given at the end. Note that smaller versions take longer to process since the absorption rate is smaller.

3.5 Rhythmic Keccak: SCA Security and Low Latency in HW

In this section, we first analyze the round-based DOM KECCAK-f implementations from [100] and point out a flaw that can possibly lead to successful attacks. We describe how careful tracing of the non-completeness property can be done to fix this flaw for any order. We show the different behavior between the original design and our proposal for first- and second-order implementations using TVLA with 55 million of traces. Then, we discuss how TI can be used for unrolled implementations without breaking the non-completeness property for the first time in the literature. Finally, we present a first-order secure low latency KECCAK-f implementation that performs an encryption in $20.61ns$ making it the fastest SCA secure implementation published to date. This work is entirely on KECCAK-f [200], which is suitable for lightweight architectures. However, benefiting from the *Matryoshka principle*, as mentioned before in Sect. 3.1.1, the concepts presented below can be relevant to other choices of b.

3.5.1 Securing Keccak with $d + 1$ masking

KECCAK has been implemented for a variety of platforms and constraints. The smallest round-based implementation claiming security against SCA is presented in [100] and uses DOM. In this section, we first show a potentially exploitable weakness of that round based-implementation due to a failure on the non-completeness property. We then verify the observability of our claims on an FPGA using t-test based leakage detection. Finally, we discuss how a sharing using $s_{in} \geq d + 1$ shares should be implemented such that it satisfies the non-completeness property and the restrictions of such an implementation.

Note that our analysis covers round-based implementations of any order but does not cover the serialized architectures presented in [100].

Analysis of DOM-Keccak

A traditional round-based implementation of KECCAK receives the input of θ from a register; θ, ρ, π, χ and ι operations are calculated within the same clock cycle and the output of ι is written to a register. When we consider a $d+1$-share implementation, it is evident that χ-refreshing needs to be separated from χ-compression by registers to achieve non-completeness marking the end of the cycle. The differences between these two architectures are depicted in Figure 3.17.

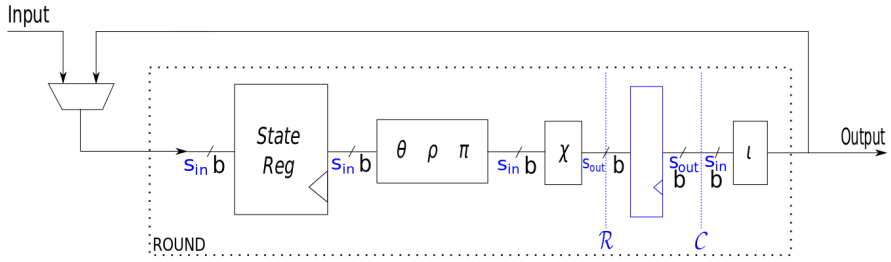


Figure 3.17: Datapath of protected (blue) and unprotected round-based Keccak implementation with and without χ-compression respectively.

The first-order implementation in [100] uses the sharing of χ in Eqn. (3.6) up to the synchronization register. The randomness r is taken from another (independent) part of the state. The synchronization is performed only on the *cross-domain* shares $S'_2[x]$ and $S'_3[x]$ by using a register triggered on the negative edge of the clock while triggering the state register in the positive edge (double clocking). Analyzing each linear and non-linear step individually as is done in [100], shows that each separate step is secure against first-order SCA. Moreover, it can easily be verified that the non-completeness property is satisfied in each separate step. However, the non-completeness of any implementation should be verified from register to register since the glitchy behavior can accumulate.

$$
\begin{aligned}
S'_1[x] &\leftarrow S_1[x] \oplus (S_1[x+1] \oplus 1) S_1[x+2], \\
S'_2[x] &\leftarrow S_1[x+1] S_2[x+2] \oplus r, \\
S'_3[x] &\leftarrow S_2[x+1] S_1[x+2] \oplus r, \\
S'_4[x] &\leftarrow S_2[x] \oplus (S_2[x+1] \oplus 1) S_2[x+2].
\end{aligned} \tag{3.6}
$$

After analyzing one round of the aforementioned implementation, we noticed that the non-completeness property from register to register is not satisfied for 112 out of 200 round output bits invalidating a condition for security. Below we trace back the input shares used to calculate a specific bit of the state particularizing Eqn. (3.1) for first-order secure implementation. A graphical representation is also provided in Fig. 3.18. We use S_i^f to refer to the output of f for share S_i.

Tracing back output bit $S^{\chi}[4,1,0]$ of the χ permutation:

$$\chi^{-1}: \quad S_3^{\chi}[4,1,0] \leftarrow S_2^{\pi}[0,1,0]\, S_1^{\pi}[1,1,0] \oplus r$$

$$\pi^{-1}: \quad S_2^{\pi}[0,1,0] \leftarrow S_2^{\rho}[3,0,0]$$

$$S_1^{\pi}[1,1,0] \leftarrow S_1^{\rho}[4,1,0]$$

$$\rho^{-1}: \quad S_2^{\rho}[3,0,0] \leftarrow S_2^{\theta}[3,0,4]$$

$$S_1^{\rho}[4,1,0] \leftarrow S_1^{\theta}[4,1,4]$$

$$\theta_1^{-1}: \quad S_2^{\theta}[3,0,4] \leftarrow S_2[3,0,4] \oplus \bigoplus_{y'=0}^{4} S_2\left[2,y',4\right] \oplus \bigoplus_{y'=0}^{4} S_2\left[4,y',3\right]$$

$$S_1^{\theta}[4,1,4] \leftarrow S_1[4,1,4] \oplus \bigoplus_{y'=0}^{4} S_1\left[3,y',4\right] \oplus \sum_{y'=0}^{4} S_1\left[0,y',3\right]$$

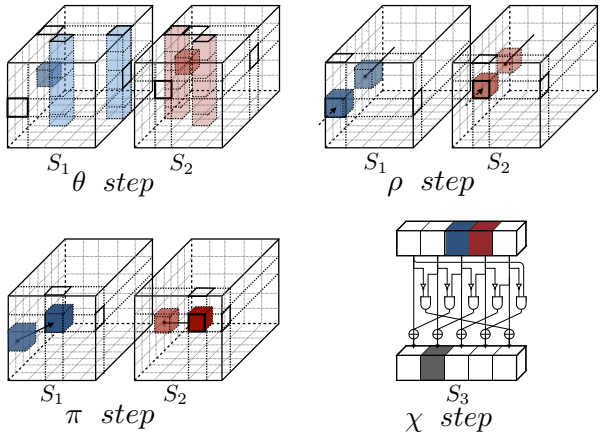


Figure 3.18: Tracing back output bit $C^{\chi}[4,1,0]$.

From these expressions it can be derived that the output bit $S_3^{\chi}[4,1,0]$ of the non-linear layer depends, among others, on the input bits $S_1[3,0,4]$ and $S_2[3,0,4]$. This means that all shares of $[3,0,4]$ are used and hence, non-completeness fails. Note that similar complete output bits can also be found for higher-order implementations. We provide a second-order example in App. A.3.

Evaluation. To illustrate the problem presented above we tested our first-order round-based DOM-KECCAK implementation that follows the structure depicted in Fig. 3.17 on FPGA. Note that we used our own implementation instead of the one provided by the authors of [100] in [167][2]. The differences between our design and that of [167] are (1) we do not use negative-edge triggering for the cross-domain shares for ease of analysis leading to a two cycle per round implementation; and (2) we always use fresh randomness to ensure that a possible problem is not caused by the degradation of uniformity. To perform the evaluation we follow the procedure presented in § 2.8. All designs are analyzed under the same conditions throughout this section.

Figure 3.19 shows results of TVLA for 20 thousand traces when masks are off and 55 million traces when masks are on for a DOM-KECCAK-f implementation. In this case, although the leakage is reduced greatly, we see that the t-score goes

[2]Please note that we have contacted the authors of [100]. They have acknowledged our findings and provided an update of their implementation accordingly [99]

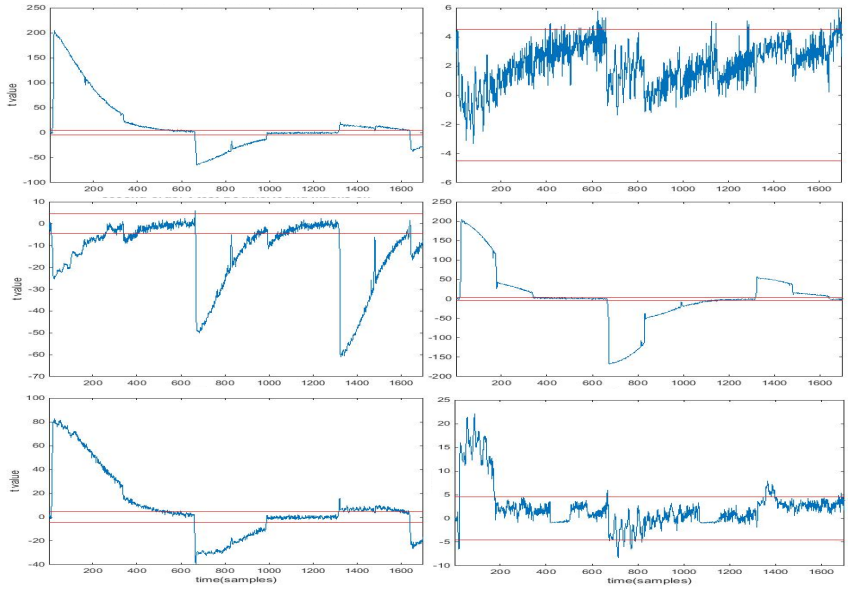


Figure 3.19: TVLA on our first-order DOM-Keccak implementation. Left: Masks off with 20k measurements. Right: masks on with 55M measurements. From top to bottom: first-, second-, and third-order analysis respectively.

over 4.5 for a significant number of sample points during both first and second rounds. We emphasize that we make no claims on this observation leading to a successful attack. Our goal is to make leakage assessment comparison between this implementation and our proposal which is presented in the next section.

Non-Complete Round-Based Architecture

The question that naturally arises after this analysis is the root of the non-completeness failure. Clearly, it is important to take into account the effect of the linear layers in combination with the non-linear layer within a cycle, since χ and the linear operations including θ are secure individually. A detailed observation shows the following.

Order of operations. The order of linear and non-linear operations within a round is important. If the round structure of a cryptographic algorithm follows the traditional SPN approach where the linear operation follows the non-linear operation within a round, we would not observe this failure of non-completeness

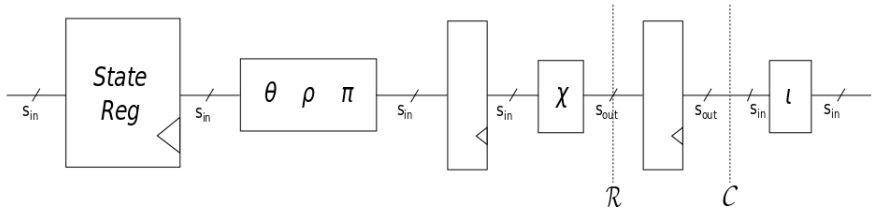


Figure 3.20: One round structure of the naive fix, with refreshing ($\mathcal{R}$) and compression ($\mathcal{C}$) before ι.

if the non-linear operation is non-complete. For higher orders, one still needs to ensure composability, which is not the objective of this work's discussion.

Linear transformations. The structure of KECCAK-f linear operations, in particular θ which shuffles and *combines* several state bits, causes the non-completeness to fail when χ is applied later. Operations ρ and π being simple wirings do not combine several bits and hence do not cause any problem.

Naive fix. Following above observations, one can clearly see that it suffices to introduce a register between the linear operations and χ to secure the design. However, this new layer of registers increases the number of cycles required per round without a significant increase in maximum frequency. Fig. 3.20 illustrates this architecture.

Our design. In order to avoid this penalty in the latency, we push the state registers to after the linear operations, using this layer of registers to break the dependencies created by these operations. In this case we need two clock cycles to perform one round and hence 36 cycles for KECCAK-f[200]. Figure 3.21 presents this structure, with which the implementation is fulfilling non-completeness. Note that it is possible to neg-edge clock the second layer of registers saving one clock cycle and reducing the area by approximately 2800GE. Here, we use two separate registers for clear analysis of the target difference.

Evaluation. We present the results for our non-complete KECCAK (NC-KECCAK) for both first- and second-order implementations, which follow the structure described in Fig. 3.21. We kept exactly the same setup as in the previous experiment for comparability. In the case of the first-order secure

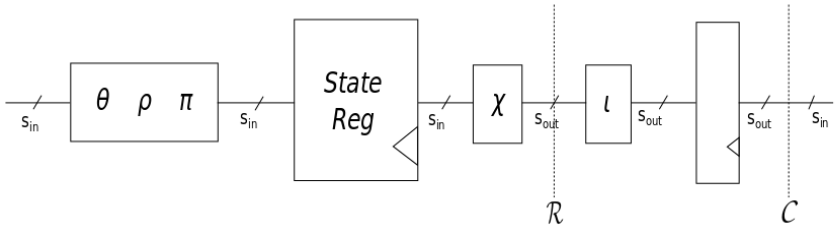


Figure 3.21: One round structure of NC-Keccak, with the state register after the linear permutations and ι before the second layer of registers.

implementation no first-order leakage appears, while in the second-order neither first- nor second-order leakage appear up to 55 million traces. Fig. 3.22 presents these results. Note that there is an outlier point in the top-right plot going over the threshold. We checked that indeed this is due to standard variations.

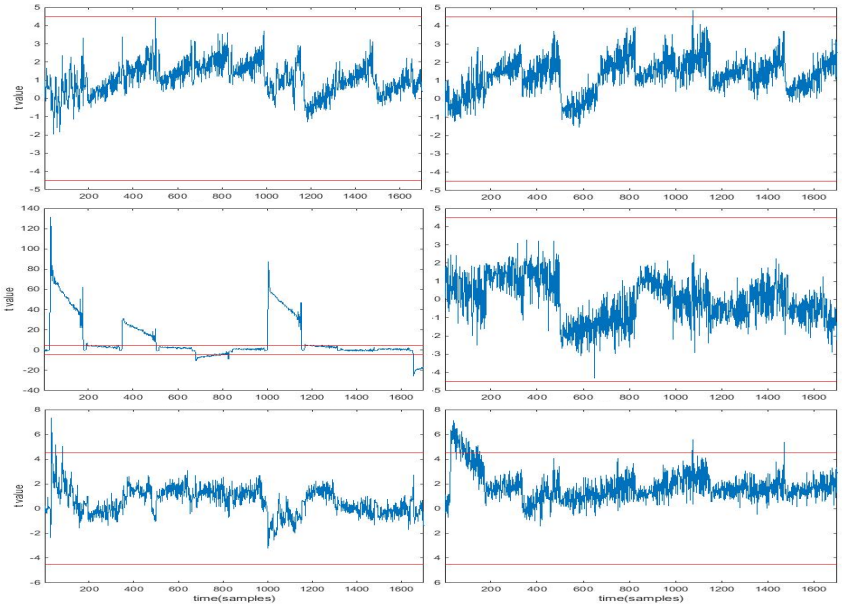


Figure 3.22: TVLA on our first (left) and second-order (right) NC-Keccak implementations over 55M traces. From top to bottom: first-, second-, and third-order analysis respectively.

Performance Analysis

Table 3.4 shows the results for this section's implementations. Note that the number of random bits presented is for each round, i.e., for every two clock cycles, and can be reduced significantly using the ideas from [38, 69, 100] which is not the goal of this work. What is important is that the performance results with and without the suggested fix are comparable.

Table 3.4: Synthesis results for the different designs

	AREA(GE)						Max.Freq.
DESIGN	χ	θ	State	Total	Rand.	Cycles	(MHz)
Plain	542.63	638.4	1 333	2 759	-	18	1 136
DOM-KECCAK 1st	9 881*	1 600	2 667	17 105	200	36	1 087
NC-KECCAK 1st	2 613	1 600	5 334*	17 493	200	36	1 300
DOM-KECCAK 2nd	23 177*	2 400	4 000	33 535	600	36	1 111
NC-KECCAK 2nd	6 200	2 400	12 001*	35 223	600	36	1 205

* This number includes the registers layer needed before the compression

3.5.2 Low Latency Masking

A typical way to reduce latency is by using an unrolled implementation where more than one round of the algorithm is implemented to finish within one cycle. Even though examples of such unprotected implementations exist, there are no such SCA protected implementations to our knowledge.

In what follows, we discuss the input and output shares on an unrolled implementation and their impact on the choice of security level in each round/layer. In order to do that, we first focus on unrolling a quadratic function twice, i.e. implementing two layers in one cycle. Then we generalize our observation to implement $F = R^N \circ \cdots \circ R^1$ with N layers, each layer being a degree t_{R^i} function, in one cycle. We derive a formula for the security order to consider in each layer (d_{R^i}) without specifying the exact sharing or any claims on optimality.

Quadratic Functions

Let R^i be a quadratic round function ($t_{R^i} = 2$), similar to that of KECCAK permutations. And let us initially target two rounds/layers, $F = R^2 \circ R^1$. Now

we want to share a quartic function ($t_F = 4$) in a cycle, so that a register in between R^2 and R^1 is not needed.

First-order resistance. For additional simplicity, let's assume R^i is composed of a series of AND gates. The dependency on the input bits for one output bit is provided in Fig. 3.23. The indices in the parenthesis describe the input shares used in each share of a particular intermediate variable given TI as the underlying sharing. It is clear that if we use a first-order secure sharing for the first layer (Fig. 3.23, left), it would indeed be secure. However, the second layer which uses the first layer's output can not satisfy non-completeness, i.e. every output share of F would depend on all input shares of F.

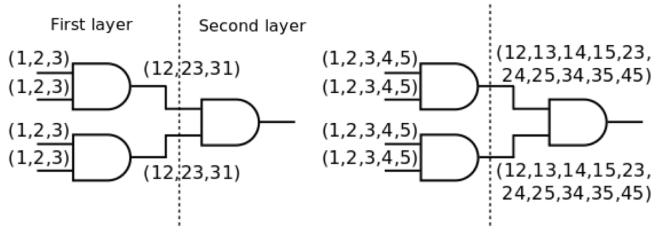


Figure 3.23: Quartic operation with (left) first-order sharing and (right) second-order sharing for the first round/layer of operations.

Since the security degrades with each non-linear layer due to combination of shares, and that we aim first-order security even at the last layer, we decided to define the security order of each layer by crawling backwards from a single output bit to input bits and observing the requirements. Moreover, we benefit from the dth-order non-completeness property. That is, we observe that first-order non-complete output bits of F requires for each combination of two input shares of R^2 to be independent of the secret. That is any combination of two output shares of R^1 should be independent of the input which is the definition of second-order non-completeness. Hence, if we implement R^1 and R^2 such that they satisfy second- and first-order non-completeness respectively, F would satisfy first-order non-completeness (Fig. 3.23, right).

Second-order resistance. We can repeat the same train of thought to analyze the security level each layer needs for second-order secure F. That is, if each combination of two output shares of R^2 is required to be independent of the secret, due to the nature of 2-input AND gate, each combination of four output shares of R^1 should be independent of the secret. This implies a

fourth-order secure implementation for the first layer whereas a second-order secure implementation for the second layer.

Linear layer. It is only natural to question the impact of linear layers between quadratic operations. If we use a TI with $td + 1$ sharing instead of a $d + 1$ DOM-indep like sharing, we have the freedom to have dependent shared inputs in each layer leading to a trivial inclusion of linear layers by simply producing the jth output shares using only j^{th} input shares. We emphasize here the importance of considering the non-completeness of the whole cycle instead of only non-linear layers.

Extending the Methodology

The above observation can be extended to higher degree functions and more rounds. Consider $F = R^N \circ \cdots \circ R^1$ where target security level for F is d_F. Similarly to the previous reasoning, we can crawl back to decide the security level desired in each layer by using the following recursive formula:

$$d_{R^i} = \begin{cases} d_F & \text{if } i = N, \\ t_{R^{i+1}} * d_{R^{i+1}} & \text{if } i < N \end{cases} \tag{3.7}$$

Note that the above recursion is independent of the sharing. It neither specifies the particular sharing to be used in each layer for all possible functions nor necessarily provides the optimal result. This formula offers a baseline for designers to systematize the securing process of unrolled implementations. It provides the security order required for the scheme to secure each layer of the unrolled function independently. Then, the designers are free to chose the sharing of their choice, provided that this sharing is secure in the required order. Moreover, when higher-order SCA resistance is considered, the particularities of the refreshing layer become critical to provide multi-variate security. The above formula does not specify the re-masking that should be used in order to provide multi-variate security for the whole algorithm.

3.5.3 Speeding Up Keccak Implementations

Our goal in this section is to push the limits of low-latency masked KECCAK-f implementation using standard CMOS-like cells. Clearly, it is challenging to provide a secure unrolled implementation of KECCAK-f that calculates more than one round in one clock cycle using a $s_{in} = d+1$ share masking scheme due

to its high diffusion per round possibly causing dependencies in shared input variables. However, TI becomes advantageous in this setting with its increased number of shares and security even with dependent shared inputs.

We first discuss following Eqn. (3.7) where TI is used as the masking scheme and s_{in} is taken as the minimum from Eqn. (2.2). We elaborate on the challenges we had and the possibility of using another TI sharing with larger s_{in} to reduce the cost and increase performance. Our security analysis and synthesis results indicate that this is the fastest protected hardware implementation published to this date.

First Attempt for Keccak-*f*

According to Eqn. (2.2), a first-order SCA resistant implementation of a quartic function requires $s_{in} \geq 5$ shares given the algebraic normal form (ANF) of this function. However, for KECCAK-f and many other algorithms alike having a non-complete and uniform sharing based on the ANF of F is not trivial due to the high diffusion of state bits. For these algorithms, having a systematic approach such as the one described in Section 3.5.2 can be useful.

Sharing R^1. Following Eqn. (3.7) we start with a second-order non-complete TI sharing with $s_{in} = 5$ and $s_{out} = 10$ ($5 \rightarrow 10$) for R^1. We provide the output dependencies on the input shares for one such sharing in Eqn. (3.8). The exact sharing, which is detailed in App. A.4.1, is based on the sharing of an AND/XOR gate introduced in [40]. It can be verified that every combination of two outputs is independent of at least one input share.

$$\begin{aligned}
S'_1 &= f_1(S_1, S_2) & S'_6 &= f_6(S_1, S_3) \\
S'_2 &= f_2(S_2, S_3) & S'_7 &= f_7(S_3, S_5) \\
S'_3 &= f_3(S_3, S_4) & S'_8 &= f_8(S_5, S_2) \\
S'_4 &= f_4(S_4, S_5) & S'_9 &= f_9(S_2, S_4) \\
S'_5 &= f_5(S_5, S_1) & S'_{10} &= f_{10}(S_4, S_1)
\end{aligned} \tag{3.8}$$

Sharing R^2. When we choose the sharing of R^2 that makes it first-order secure, we also need to ensure that the non-completeness holds when R^1 and

R^2 are combined. Due to the linear layer of KECCAK-f, the AND gates in R^2 depend on the outputs of more than two AND gates of R^1. Hence, not every first-order non-complete sharing of R^2 leads to a non-complete F. Moreover, even though many direct $10 \rightarrow 10$ sharings of R^2 satisfying the above restriction can be derived following a simple procedure, clearly we prefer a compatible first-order secure non-complete sharing for R^2 where $s_{in} = 10$ and $s_{out} = 5$ ($10 \rightarrow 5$) in order to avoid further increase in number of shares. Hereon, we call a sharing where $s_{in} > s_{out}$ a *compression sharing*. Note other other examples of compression sharings can be found in the literature [39, 41].

For a target of $10 \rightarrow 5$ compression sharing, we need to be careful since many shares are combined. Eqn. (3.9) illustrates the input-output dependency of one possible sharing for $\mathcal{R}^2$. We provide the exact sharing we use in App. A.4.2.

$$\begin{aligned}
S''_1 &= f_1(S'_1, S'_2, S'_3, S'_6, S'_9, S'_{10}) \quad \text{(Missing input } S_5\text{)} \\
S''_2 &= f_2(S'_1, S'_2, S'_5, S'_6, S'_7, S'_8) \quad \text{(Missing input } S_4\text{)} \\
S''_3 &= f_3(S'_1, S'_4, S'_5, S'_8, S'_9, S'_{10}) \quad \text{(Missing input } S_3\text{)} \\
S''_4 &= f_4(S'_3, S'_4, S'_5, S'_6, S'_7, S'_{10}) \quad \text{(Missing input } S_2\text{)} \\
S''_5 &= f_5(S'_2, S'_3, S'_4, S'_7, S'_8, S'_9) \quad \text{(Missing input } S_1\text{)}
\end{aligned} \tag{3.9}$$

Our Final Design

Optimized Sharings. The greatest area contributor of a KECCAK-f parallel implementation, is the χ step. The number of gates in the non-linear operation scales with the number of input shares as follows:

- Number of AND gates $= \binom{s_{in}}{2} + s_{in}$
- Number of XOR gates $= \binom{s_{in}}{2} + s_{in}$
- Number of NOT gates $= s_{in}$

The area overhead of R^2 caused by increasing the number of shares to 10 is significant. Therefore, we decided to look for another sharing for the first layer that produces less outputs. One possible option would be the $6 \rightarrow 7$

sharing proposed in [40]. On the other hand, by investigating further we found $6 \rightarrow 6$ sharings for both layers which provides first-order non-completeness when combined. The abstract sharings and the exact equations are provided in Eqn. (3.10), and in App. A.4.3.

$$
\begin{aligned}
S'_1 &= f_1(S_1, S_2, S_3) & S''_1 &= f_1(S'_4, S'_5, S'_6) & &\text{(Missing input } S_1\text{)} \\
S'_2 &= f_2(S_1, S_4, S_5) & S''_2 &= f_2(S'_2, S'_3, S'_5) & &\text{(Missing input } S_2\text{)} \\
S'_3 &= f_3(S_1, S_4, S_6) & S''_3 &= f_3(S'_2, S'_3, S'_4) & &\text{(Missing input } S_3\text{)} \\
S'_4 &= f_4(S_2, S_5, S_6) & S''_4 &= f_4(S'_1, S'_4, S'_5) & &\text{(Missing input } S_4\text{)} \\
S'_5 &= f_5(S_3, S_5, S_6) & S''_5 &= f_5(S'_1, S'_3, S'_6) & &\text{(Missing input } S_5\text{)} \\
S'_6 &= f_6(S_2, S_3, S_4) & S''_6 &= f_6(S'_1, S'_2, S'_6) & &\text{(Missing input } S_6\text{)}
\end{aligned}
\tag{3.10}
$$

With this sharing we are able to reduce the area considerably without failing non-completeness.

Performance Analysis. We present the performance results for the two round unrolled implementation using the sharings presented in Sections 3.5.3 and 3.5.3 in Table 3.5 for comparison.

Table 3.5: KECCAK-f[200] first-order secure unrolled implementations' performance results using NanGate 45nm library. Comparison with Parallel implementations from [38] and PARALLEL double clocked and PARALLEL pipelined from [99]

	AREA(kGE)							Max.Freq.	Latency
DESIGN	χ^1	χ^2	θ^1	θ^2	State	Total	Cycles	(MHz)	(ns)
$5 \rightarrow 10 \rightarrow 5$	13.53	59.89	4.28	9.75	6.7	99.34	9	395.25	22.77
$6 \rightarrow 6 \rightarrow 6$	22.40	22.16	5.66	5.66	8.03	70.12	9	436.7	**20.61**
Previous work*									
3sh [38]	-	-	-	-	27.2	116.6	25	592	42.23
4sh [38]	-	-	-	-	36.3	139.4	24	588	40.82
D.c. [99]**	-	-	-	-	38.9	100.5	48	803.9	59.71
Pipel. [99]**	-	-	-	-	36.8	111.8	72	837.5	85.97

* Implementations of KECCAK-f[1600]
** Results gathered with library UMC 130nm

The cost of χ^1 and θ^1 indicates that R^1 in $5 \rightarrow 10 \rightarrow 5$ sharing is a bit cheaper than the one using $6 \rightarrow 6 \rightarrow 6$. However, χ^2 of $5 \rightarrow 10 \rightarrow 5$ is significantly more expensive due to the greater number of inputs as expected. In the $6 \rightarrow 6 \rightarrow 6$ scheme the state register is a bit bigger since there is one more share to store. All things considered, the area reduction of our design compared to a first attempt is clear.

Given the results, it is possible to see that the time needed for a single encryption is $20.61ns$ when synthesizing the design with the NanGate 45nm library.

Evaluation. We evaluate the optimized design using the $6 \rightarrow 6 \rightarrow 6$ sharing. The test is done under the conditions already presented in Section 3.5.1. Since this implementation is much faster than the previous ones, the analysis covers 10 rounds of the cipher.

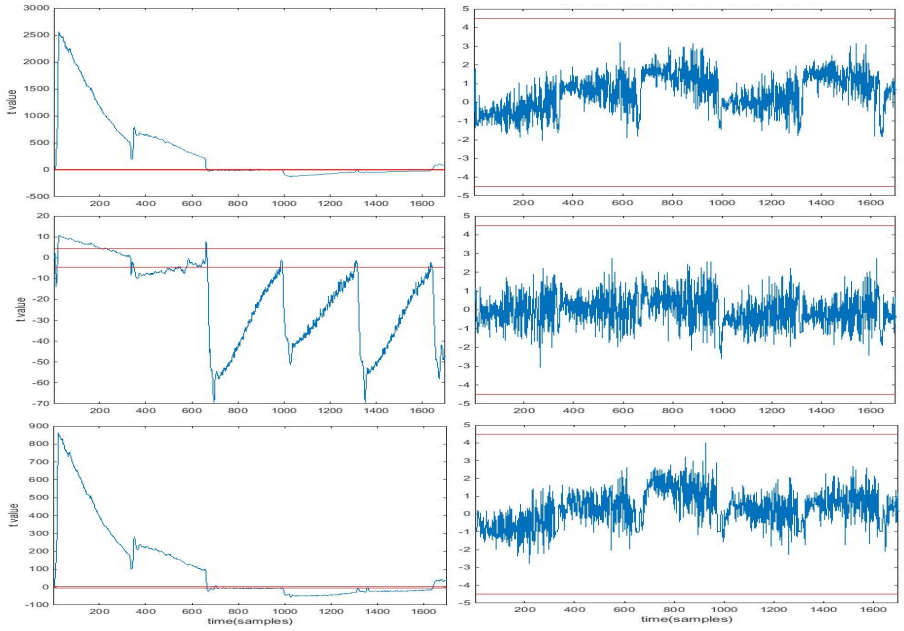


Figure 3.24: TVLA on first-order secure double round KECCAK implementation. Left: Masks off. Right: masks on. From top to bottom: first-, second- and third- order analysis.

The results are presented in Fig. 3.24 for masks off and masks on. It is possible to appreciate that the countermeasures indeed prevent leakage from appearing in the first-order. Moreover, we observe no leakage in the second or in the

third order t-test using 55 million measurements. We attribute this to the noise introduced by the large number of shares used. Therefore, we can affirm this is the fastest first-order SCA resistant KECCAK implementation published to this date.

3.6 Conclusions

In this chapter we have presented several implementations, both unprotected and SCA protected, of some of the KECCAK family algorithms. These algorithms include the KECCAK-f permutations, the Authenticated Encryption scheme KETJE, and the SHA-3 standard. Via these implementations, we have explored different concepts that are crucial or may help the designer to secure implementations against Side-Channel Analysis. We have also analyzed the advantages and disadvantages of different masking schemes, and the trade-offs they entail, related to area, latency, and randomness.

We have analyzed in detail the importance of independent inputs to correctly fulfill non-completeness when applying masking, and the effect on different examples when this is not provided. Furthermore, we found a flaw related to this concept on one of the state-of-the-art round-based implementations of KECCAK, demonstrating how intricate the shares dependencies are. We proposed a new implementation fixing the problem, where a new layer of registers is added in the right place to synchronize the signals in order to comply with the non-completeness property, which we tested in practice to confirm our claims.

Then, we explored the use of the "Changing of the Guards" technique to help in achieving uniform sharings with lower number of shares when using TI, with the implementation of a protected KETJE for encryption and decryption using this technique. This helped us to reduce the number of shares needed to secure the KECCAK-f S-Box from four to three when using a TI sharing scheme, considerably reducing the area requirements while still satisfying uniformity.

Finally, based on the investigation on the different sharing schemes, we developed a generic method for securing unrolled implementations for low-latency applications using TI. As a proof-of-concept, we applied this method to design a first-order SCA secured KECCAK implementation. Moreover, we evaluated the design's security in practice to see that no first- and neither second-order leakage was found.

Chapter 4

Glitch-Resistant Masking Schemes as Countermeasure Against Fault Sensitivity Analysis

On our transition towards incorporating FA protection into our designs, we analyze the effect of another important attack, proposed almost a decade back, known as Fault Sensitivity Analysis (FSA) [129]. We consider this attack as a hybrid attack since a fault is injected to produce a side-channel, which is exploited with SCA techniques. Hence, we investigate the performance of Threshold Implementations against this kind of attack, to see if just masking alone can thwart it, and what are the requirements.

In this chapter, we show both theoretically and practically that glitch-resistant masking schemes, such as TI, provide additional protection against Fault Sensitivity Analysis (FSA). We ground this observation in several experiments that involve both ASIC and FPGA implementation. We simulate FSA attacks on different unprotected and SCA protected S-Boxes to show their resilience against FSA. Attacks on the glitch-resistant masked implementations fail even in a scenario that is beyond worst-case in practice: an attacker with advantageous noiseless, high-precision, and simulated observations does not succeed in retrieving the key.

This chapter is based on the paper published in Fault Diagnosis and Tolerance

in Cryptograhy (FDTC) 2018 [10]. The original idea comes from Thomas De Cnudde. My contribution in this work was to provide additional theoretical reasoning of why TI prevents FSA, devise the experiments to practically prove it, and perform the FPGA experiments.

4.1 Fault Sensitivity Analysis

FSA is an actively triggered side-channel attack (also known as hybrid attack) that can retrieve secret keys from the correlation of processed data values with their Fault Sensitivity (FS) [129]. The fault sensitivity is the intensity of an injected fault at which the device shows a detectable characteristic, e.g. supply voltage and/or clock frequency where the device starts outputting incorrect results.

An FSA attack is mounted in two phases: a collection or profiling phase, and a key retrieval phase. In the profiling phase, fault sensitivities of input values are collected. The FS of a value is retrieved by repeatedly encrypting a fixed plaintext. During each encryption, the intensity of the applied fault is gradually increased. The first fault intensity at which the output becomes faulty is noted as the fault sensitivity for that input. In the case of clock glitching, this corresponds to shortening the clock period momentarily to a value lower than the propagation delay caused by the input transition. This process of retrieving the FS is repeated for several different plaintexts. In the subsequent key retrieval phase, the FS information is used to find the highest correlation between the predicted FS from a key guess and the measured FS. The highest correlation then leads to the correct key [129].

Two factors that codetermine the success of an FSA attack are the resolution of the fault injection the attacker can apply, and whether or not a model for the FS and the processed data can be extracted. A clock glitch generator has shown to provide enough resolution for successfully mounting FSA attacks on both Integrated Circuit (IC) and Field Programmable Gate Array (FPGA) implementations [85, 86]. The Hamming Weight model has been used with success [129], but was shown to not work on all S-Boxes [135]. An alternative model based on the zero-value attack was shown successful in the latter cases [136].

4.2 Related Work

In the original work of Li, it was argued that masking schemes could provide resistance against FSA [129]. After several masking schemes were broken using FSA [136, 139, 140], dedicated countermeasures against FSA have been proposed. These can be categorized into gate-level countermeasures and RTL-level countermeasures.

The first gate-level countermeasure against FSA was proposed by Ghalaty et al. [92]. Their proposed method is based on internally changing the circuit in order to balance out the sources of FSA leakage with low area overhead.

Another approach to counteracting FSA attacks is through RTL-level countermeasures [128]. By only allowing the results of the combinational logic to the inputs of the capturing register after its outputs have been stabilized using an enable signal, the fault sensitivity is made constant and equal to the arrival of the enable signal. This method was shown possible by Endo et al. by choosing the enable signal to be greater than the largest critical path delay of the circuit during post-manufacturing reconfiguration [83, 84].

While these countermeasures have a reasonably small impact on the area and throughput, the resulting circuits are still vulnerable to passive SCA. It is reasonable to expect an integrated circuit to require both protection against SCA and FAs and therefore, combinations of Threshold Implementations and FA countermeasures have been investigated in more recent works: ParTI [168] and Private Circuits-II enhanced TI [64]. Since the latter two approaches build on the properties of TI, we believe the claims of FSA resistance we make in this work can directly be applied to them.

4.3 The power of SCA glitch-resistance against FSA

We provide theoretical insights on why glitch-resistant schemes against SCA are also a valid protection against Fault Sensitivity Analysis. We begin by giving the assumptions we make on the attacker, and proceed by explaining the concepts that lead to FSA resistance.

4.3.1 Assumptions

In order to facilitate our explanation, we make the following assumptions:

1. we assume that the attacker accurately measures the timing properties of the circuit, i.e. the propagation delay through his FSA attack,

2. we assume the players to leak independently and that the requirements for the masking schemes are satisfied.

The validity of the first assumption can be explained by looking at the physical effect of the FSA attack. The attacker applies a fault and gradually increases its intensity, each time noting whether the output was correct or incorrect. After m injections, the attacker holds a list of m responses $[(\Delta_1, FI_1), ..., (\Delta_m, FI_m)]$ [135], symbolizing whether or not the fault injection led to correct or incorrect outputs under fault intensity FI_i. If $\Delta_i = 1$, we consider the computation to have finished correctly under fault intensity FI_i and $\Delta_i = 0$ otherwise. The fault intensity $\overline{FI}$, for which $FI_i = \overline{FI}$ leads to $\Delta_i = 1$, and F_{i+1} leads to $\Delta_i = 0$, is defined as the fault sensitivity for input x. This point can be considered as the last point in time for which not all output bits were valid yet, i.e. the critical propagation delay.

The independent leakage requirement of the second assumption means that, from a side channel point of view, the power consumption of the n different players of the masking scheme are independent from each other. In case the players operate in a parallel way (such as e.g. in TI), this means the power consumption of the players can be linearly decomposed into independent instantaneous power consumption traces. When the players operate in a serial way (such as e.g. in the implementations of the Roche and Prouff masking scheme [63, 138]), the instantaneous power consumption of the n players should not overlap in time. The assumption that the requirements of the masking scheme are satisfied is straightforward, as otherwise even the side-channel security can not be guaranteed.

4.3.2 On Non-Completeness

As pointed out in [129], signal transitions in a device are data-dependent, and thus the fault sensitivities as well. Following the same principle as the previous work uses to prove the data-dependency of the signal timing delay, we prove that masking schemes fulfilling the non-completeness property are inherently secure against FSA.

To do this, we use the following example of a TI shared AND gate. Given the shares of the two inputs x and y as (x_1, x_2, x_3) and (y_1, y_2, y_3), then the shares (z_1, z_2, z_3) of the output z are computed as:

$$
\begin{aligned}
z_1 &= f_1(x_1, x_2, y_1, y_2) = x_1y_1 + x_1y_2 + x_2y_1 \ , \\
z_2 &= f_2(x_2, x_3, y_2, y_3) = x_2y_2 + x_2y_3 + x_3y_2 \ , \\
z_3 &= f_3(x_1, x_3, y_1, y_3) = x_3y_3 + x_3y_1 + x_1y_3 \ .
\end{aligned} \tag{4.1}
$$

Fig. 4.1 presents a generic scheme of a three shares Threshold Implementation of a non-linear operation. Any TI implementation follows a similar structure, which means the claims presented below can be generalized.

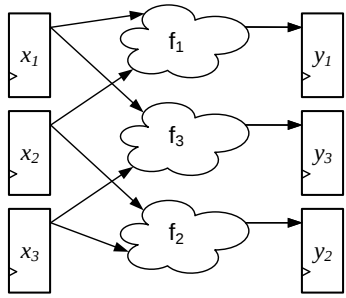


Figure 4.1: General structure of a Threshold Implementation

All three combinational circuits are computed in parallel and independently, as stated in assumption 2, where every one of them has a different propagation delay. Thus, the fault sensitivity targets solely the circuit with the longest (critical) propagation delay. Let us assume this holds for the first combinational circuit:

$$FS_1(x_{1,2}, y_1) \geq FS_2(x_{2,3}, y_2) \geq FS_3(x_{1,3}, y_1) \tag{4.2}$$

We analyze in detail the data-dependency of the transitions of the first share from the previous example, looking at the timing delay of the signals. This is depicted in Fig. 4.2, where T_c represents the timing delay of a component c, either wire or gate, and, in case of a fanout wire, T^i refers to the i-th branch.

As shown in [129], the propagation delay of a transition in an XOR gate is determined by the longest propagation delay at its inputs, independently of the value of the inputs. The authors also show that the propagation delay of an AND gate is predisposed by the values on its inputs before and after each

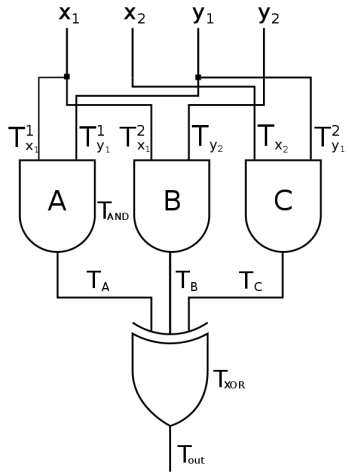


Figure 4.2: First share combinational circuit from the 3-share AND gate en Eqn. (4.1)

transition. For example, arrival of the first logic *zero* at the inputs determines the output value, hence the propagation delay, regardless of the other input bits. On the contrary, setting one of the inputs to a logic *one* can not determine the propagation delay through the AND gate until the rest of the inputs have been updated. Therefore, regardless of the initial (reset) value of the AND gate the more logic zeros the new input contains, the shorter the propagation delay. In other words, for each input transition, the propagation delay is directly correlated with the Hamming Weight (HW) of the inputs. Reversely, OR gates have the logic *one* as the dominant signal, which leads to the reverse but dual correlation between the two.

We assume the delay of any share of the variable x to be greater than the ones from y. Therefore, we can express the timing delays of the outputs of the AND gates from Fig. 4.2 as follows.

$$T_A = \begin{cases} T^1_{y_1} + T_{AND} \text{ (if } y_1 = \text{'0'}) \\ T^1_{x_1} + T_{AND} \text{ (if } y_1 = \text{'1'}) \end{cases}$$

$$T_B = \begin{cases} T_{y_2} + T_{AND} \text{ (if } y_2 = \text{'0'}) \\ T^2_{x_1} + T_{AND} \text{ (if } y_2 = \text{'1'}) \end{cases} \tag{4.3}$$

$$T_C = \begin{cases} T^2_{y_1} + T_{AND} \text{ (if } y_1 = \text{'0'}) \\ T_{x_2} + T_{AND} \text{ (if } y_1 = \text{'1'}) \end{cases}$$

It is straightforward to see that the side-channel information leaked from attacking the first combinational logic cloud only reveals information on two out of the three shares of the secret (x_1, x_2 and y_1, y_2). FSA targets exclusively the longest propagation delay, which means that it is not possible to get additional information from a different combinational cloud and combine it with previous information to retrieve the secret.

Hence, we can conclude that any Threshold Implementation fulfilling the *non-completeness* property is secure against FSA. Both the CMS and DOM sharing schemes inherit non-completeness to provide security in the presence of glitches, which means that these schemes are also secure against FSA.

4.3.3 On Uniformity

It is important to note that the example above does not have a uniform output sharing. This property is needed to provide first-order SCA security when the output is subsequently used in a non-linear computation. We emphasize that, as demonstrated above, it is not needed to provide FSA security. For a scheme to provide security against FSA it suffices that the *non-completeness* property is satisfied. It is certainly advantageous to use a uniform sharing for larger systems so that the scheme provides security against both FSA and first-order SCA.

4.4 Experiments

To illustrate the theoretical insights of our claims, we simulate the actual FSA attack using two representative TI implementations of S-Boxes.

We rely on post place-and-route simulations, in both ASIC and FPGA. This way we are able to obtain perfectly aligned noiseless observations with a timing precision of 1ps. As we extract the data-dependent propagation delays from the simulated data, we do not have to take into account any jitter introduced by the glitching apparatus. Furthermore, we observe target circuits in isolation from any other components by directly calculating the propagation delay of this circuit for a given input. Therefore we are arming our adversary with unrealistically clean measurements that are likely unattainable in practice. For the ASIC experiments, the design flow automation and data processing is done using the CASCADE [179] framework. For the FPGA experiments, we use the Isim environment to perform the simulations.

Profiling phase

In the seminal paper Li et al. [129] use the value zero as the reset value between two fault injections. We leverage the small size of our target circuits, and the high speed and high precision of our simulated approach, to show that this is not necessarily the best value to start the attack from. Instead of only collecting fault sensitivities for up to 2^N input values from an N-bit circuit, we simulate all possible $2^{2 \cdot N}$ transitions as well. From there we differentiate 2^N profiles, using all 2^N reset values. Consequently, each profile has 2^N traces. The double exponential complexity of this approach makes it infeasible for circuits with a larger number of inputs. From this approach we obtain profiles with the maximal correlation between fault sensitivities and input data, i.e. profiles which are optimal for the attacker. We then proceed to key recovery using the most effective profile.

Key Recovery

We attack a dummy cipher round consisting of a single S-Box and key addition, as depicted in Fig. 4.3. As in the profiling phase, the small input space of our targets allows us to collect all possible ciphertexts. Note that since the delay of XOR gates is not data dependent, as pointed out in [129], the XOR circuit need not be included in the profiling.

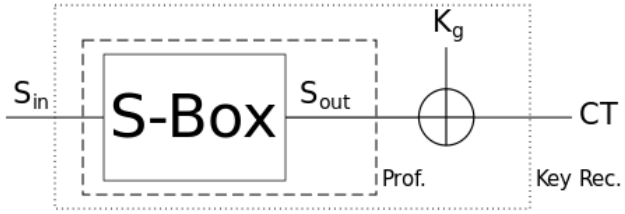


Figure 4.3: Circuit under attack, unprotected.

The profiling is made over the inputs of the S-Box, so we attack the inputs of the circuit. Eqn. (4.4) presents the function used to get the predicted fault sensitivities (FS_g), using the ciphertext (CT) and the key guess (K_g):

$$FS_g = HW(SBox^{-1}(CT \oplus K_g)) \ . \tag{4.4}$$

The guessed fault sensitivities are subsequently correlated with the observed ones, where the maximum correlation should correspond with to the correct key.

For the protected implementations, the circuit under attack has similar structure, in this case shared over three shares, as shown in Fig. 4.4. We allow the attacker to only observe unshared plaintext-ciphertext pairs, i.e. the attacker sees the dotted domain as a black box. On the contrary, we allow the attacker to see the exact sharing to do the profile, i.e. what is outside of the dashed box. This means that the attacker is able to get the exact shared input that corresponds to the profile with the highest correlation. The aim of these experiments is to immerse our S-Boxes into the worst case scenario, to test how the proposed countermeasure responds. In reality, attackers are very unlikely to attain such detailed profiling nor have such a stripped down target circuit. Moreover, the attackers would not be able to cover the whole state of inputs in a real design, which means that the attacks are limited to the amount of measurements they can procure.

4.5 Results

In this section we present the results of our experiments. We perform our experiments in both ASIC and FPGA to get a deeper and broader view of the effects of fault sensitivity analysis. As expected, the experiments show that the attacks fail against the TI implementations, even given the highly favorable

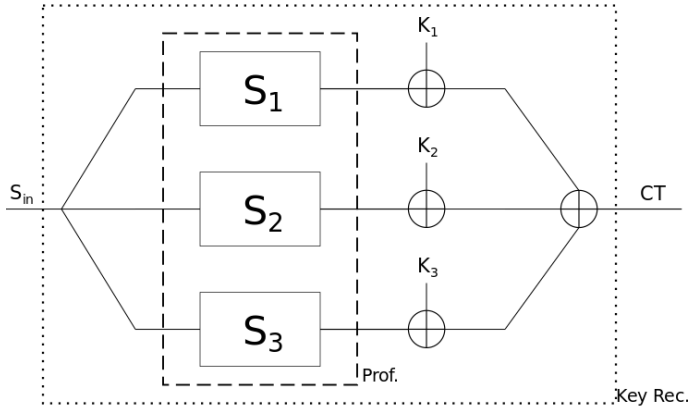


Figure 4.4: Circuit under attack, protected.

conditions for the attack. We present results for two target circuits that illustrate different scenarios: FSA resistance with a conventional TI implementation, and FSA resistance with a non-uniform implementation.

Present

Our first target is the 4-bit S-Box of the lightweight ISO standard PRESENT blockcipher. We do a preliminary analysis on the PRESENT S-Box due to the smaller size compared to the KECCAK S-Box, in order to provide exhaustive results.

For the protected design we use the TI approach from Poschmann et al. [151]. This design is decomposed into two quadratic S-boxes F and G forming a pipelined implementation. As already mentioned before, FSA targets the longest propagation delay. To comply with this, we target just one of the two stages to gather the profiles, choosing the first stage F for our experiments. The unprotected implementation has a total of 2^4 inputs and $2^{2\cdot4}$ possible transitions that can occur on the inputs. The protected implementation is split into three shares per variable, in accordance with the TI principles. The total number of inputs (resp. outputs) is thus 12 (4 sensitive bits masked with 3 shares), resulting in the total of $2^{2\cdot12}$ transitions.

Fig. 4.5 shows correlation values between the HW of the input and the FS for profiles with different reset values. Prominent correlation peaks appear in the case of the unprotected implementation. In case of ASIC as target, the correlation peaks at 0.75 for reset value 0b1011, whereas in the case of FPGA

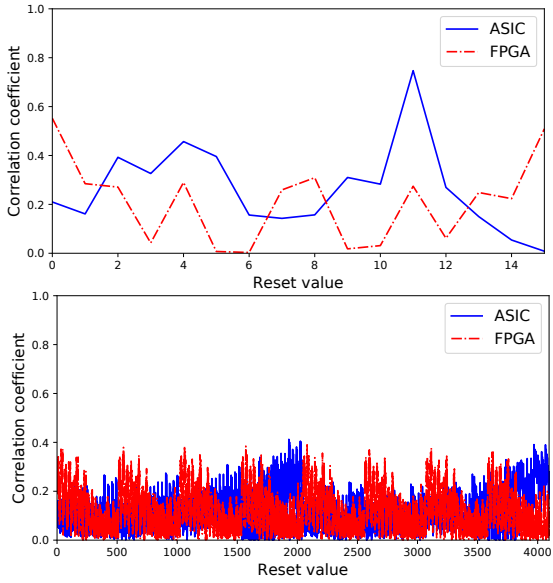


Figure 4.5: ASIC vs FPGA correlations for all profiles of the unprotected (top) and protected (bottom) PRESENT S-Box.

as target, peak value is as high as 0.55 corresponding to reset value 0b0000. It is interesting to see that this peak is closely followed by the 0.51 value for reset value 0b1111. We will discuss this in the later Sec. 4.6.

In case of the protected implementation, the attacker is allowed to profile the S-Box for all 2^{12} shared reset values to maximize the attacking precision. Still, correlation peaks do not surpass 0.41 for ASIC with reset value 0x798, nor 0.39 for FPGA with reset value 0x821. In comparison, the correlation values corresponding to the all-zero reset vector are 0.08 for ASIC and 0.24 for FPGA. This demonstrates that the all-zero reset (or initial state) vector may be an effective one, but it is not necessarily the best one.

At first glance it would stand to reason to maximize the number of collected traces by looking at the correlation between the HW and the FS across all transitions. Nevertheless, doing so results in consistently lower correlation values across all of our experiments. Fig. 4.6 provides a graphical explanation of this.

Similarly, we tried using the correlation of Hamming Distance (HD) and FS in both scenarios, only to obtain significantly lower correlation values. We attribute this to the nature of the side channel we attempt to exploit compared

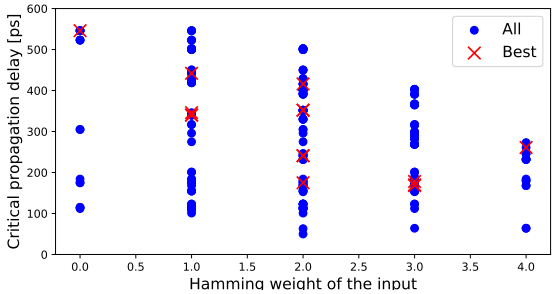


Figure 4.6: ASIC FS of the unprotected PRESENT S-Box for the highest correlation profile.

to the classical SCA. Let us observe a 2-input AND gate, transitioning first 0b00→0b11, and then back 0b11→0b00. The power consumption of the two transitions will normally be comparable as it is proportional to the amount of current needed to charge and discharge the circuit's capacitances. For FSA however, the differences in the input-dependent propagation delays matter. Hence a transition ending in 0b00 will on average be faster in an AND gate [129]. As a consequence of this discrepancy, the HW of the input value at the end of a transition is the most relevant observation as opposed to the HD.

Fig.4.7 illustrates the key recovery attacks on the unprotected and protected implementations. We can see that the guessed value (×) coincides with the target value (■) of the key for the unprotected implementation, representing a successful attack. This is not the case for the TI implementation, where even though the attacker is given the best reset value and unrealistically accurate profiling, correlations for each key guess remain discouraging small.

Keccak

Our second target is the 5-bit KECCAK χ permutation. We use the TI implementation from [69] for our experiments. It has three 5-bit shares along with a 4-bit input for randomness. By fixing the random bits to 0b0000 we disable the uniformity at the output of this implementation. We thereby test the claim that the non-completeness is sufficient to provide security against FSA attacks, as opposed to both non-completeness and a uniformly shared output. A qualitative difference compared to the previous experiments is that we do not exhaust all of the $2^{2\cdot3\cdot5}$ input transitions. Instead we pick $2^{2\cdot12}$ random transitions to match the previous experiment in volume. Note that the results we obtain are consistent over different pools of random inputs.

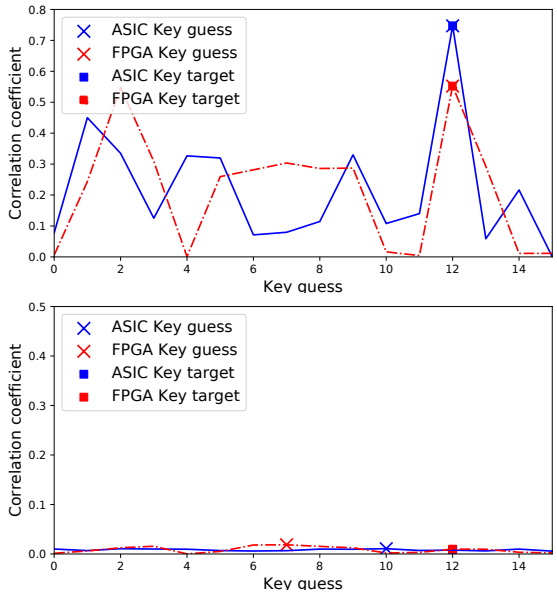


Figure 4.7: FSA attack on the unprotected (top), and TI (bottom) implementation of the PRESENT S-Box.

The results of the FSA attack are shown in Fig. 4.8, which shows that despite the lack of uniformity at the output of the TI, the non-completeness is sufficient to provide protection against FSA.

4.6 Discussions

4.6.1 The Roche and Prouff Scheme

The Roche and Prouff masking scheme [153, 162] offers an alternative approach at masking hardware implementations at any order d in the presence of glitches. Shamir's secret sharing scheme [172] is used to share the sensitive variables, and computations on the resulting shares are performed using the BGW secure multi-party computation protocol [28].

The protocol for a multiplication (or AND gate) is split into two stages to achieve glitch-resistance [138]. In neither of the two stages is there an intermediate variable that depends on any unmasked inputs, which is exactly the definition of non-completeness. Thus, we argue that this sharing scheme is also resistant

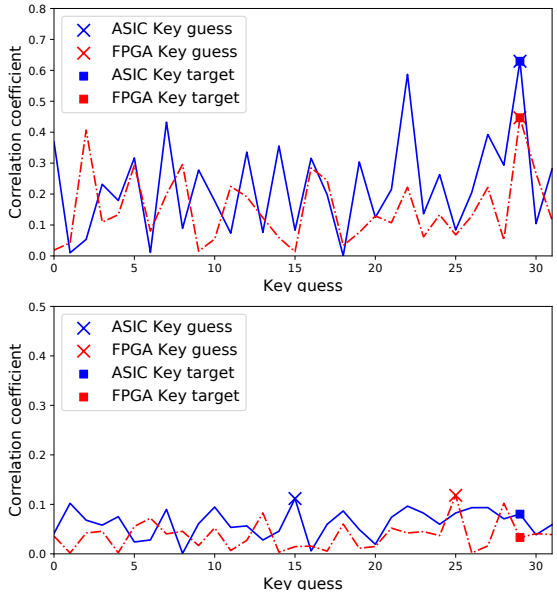


Figure 4.8: FSA attack on the unprotected (top), and TI (bottom) implementation of the KECCAK S-Box.

against FSA given that non-completeness is fulfilled. By extension, any scheme that is based on a glitch-resistant masking scheme benefits from FSA resistance, including the schemes by Reparaz et al. [158] and Seker et al. [170].

4.6.2 ASIC vs FPGA

We simulated attacks on two substantially different platforms, namely a standard-cell library for ASIC development, and an FPGA board. Albeit their difference, all obtained results are consistent across the target circuits. The first difference is that the propagation delays in FPGA are much larger. For the ASIC example, Fig. 4.6 shows that the propagation delays of the unprotected PRESENT S-Box are spread between 100–600ps. The same circuit on the FPGA leads to propagation delays that are spread between 8200–9200ps. An older technology and a higher complexity of the configurable LUTs that realize logic on an FPGA lead to this absolute difference.

A second difference stems from the 5to1 (5-input, 1-output) LUTs of the target FPGA, which allow merging multiple logic gates, leading to a lower number of LUTs than standard cells needed for the same circuit. The fine-grained standard

cells of ASICs lead to a higher spread in propagation delays. Hence, a finer distinction can be made in the latter between different input values, resulting in higher correlation peaks.

Furthermore, as depicted in Fig. 4.5, the choice of the best profile, i.e. the best reset value, depends on the target platform. In our experiments, where the target circuit is small enough to fit into one FPGA LUT, choosing an all-zero, or all-one vector as the reset value consistently leads to the best profile. For ASIC circuits, which rely on less flexible atomic cells to implement combinational logic, the best profile greatly differs from design to design. So much that even for the same RTL code synthesized under a different set of constraints, the best reset value would change.

4.6.3 Reset Value

In the original work, Li et al. [129] use an all-zero vector as the reset value. Although this is a natural choice, and coincidentally often fruitful for the FPGA implementations, we have shown that it is not necessarily the best one. In our experiments we have seen that even when attacking the unprotected implementations, the key can not be recovered for all reset values. From the perspective of a practical attacker, choosing an arbitrary reset value is not trivial and would at best require the application of a well chosen input to achieve the desired initial state of the circuit. On top of that, when attacking a real life design it is not possible to exhaust all input transitions in search of the best reset value. Therefore, by choosing the best reset value, we are yet again giving advantage to the attackers, only to see them fail.

4.7 Conclusions

In this chapter, we investigated the resistance of masking schemes against Fault Sensitivity Analysis attacks. We proved both theoretically and experimentally that glitch-resistant masking schemes featuring the non-completeness property are secure against FSA, even under extremely favorable conditions for an attacker. We strengthened our arguments through simulations in both ASIC and FPGA environments.

Finally, we conclude that the fulfillment of the non-completeness property suffices for an implementation to be secure against Fault Sensitivity Analysis (FSA), provided that the different shares leak independently.

Chapter 5

Design of SCA and FA Combined Countermeasures

In this chapter, we introduce a novel adversary model for combined attacks, and then, two different combined countermeasures against SCA and FA. The first one, named CAPA, provides provable security even in a combined attack scenario under a robust adversary. The second one, M&M, protects against both SCA and FA in a more relaxed attacker model, where security against combined attacks cannot be proven.

First, we introduce the Tile-Probe-and-Fault model, a model generalizing the wire-probe model of Ishai et al. [108]. We extend it to cover more comprehensive side-channel leakage scenarios on a chip, and also to cover fault and combined attacks. Second, we introduce CAPA: a combined Countermeasure Against Physical Attacks. Our countermeasure is motivated by our model and aims to provide security against higher-order SCA, multiple-shot FA, and combined attacks. CAPA draws much inspiration from the MPC protocol SPDZ [72] to provide provable security against combined attacks. To demonstrate that the model, and the CAPA countermeasure, are not just theoretical constructions, but could also serve to build practical countermeasures, we present initial experiments of proof-of-concept designs using the CAPA methodology. Namely, a hardware implementation of the KATAN and AES block ciphers. We verify experimentally that the design can resist second-order DPA attacks, even when the attacker can acquire millions of traces. These countermeasures are highly expensive, due to the potent attacker model proposed, perhaps overkill for most realistic scenarios.

Third, as a follow-up work from CAPA, the second combined countermeasure is presented, known as M&M: Masks and Macs against Physical attacks. We present a new fault model, derived from the wire-probe model and the newly introduced Tile-Probe-and-Fault, to create a more efficient scheme more suitable for practical scenarios. M&M combines Masking with information-theoretic MAC tags and infective computation. It works in a stronger adversary model than the existing scheme ParTI [168], yet it is a lot less costly to implement than the provably secure MPC-based scheme CAPA. As opposed to error-detecting codes, the MAC mapping is utterly unpredictable, eliminating the possibility of *smart* undetectable faults. This makes M&M secure against stronger adversaries than when using error-detecting codes. We demonstrate M&M with an SCA- and DFA-secure implementation of the AES block cipher. We perform an SCA evaluation of our implementations where no leakage is found with up to 100 million traces. Additionally, we design and perform a fault evaluation to confirm our theoretically claimed fault coverage.

Finally, we elaborate on the security against SCA, FA, and combined attacks for both schemes, and add further discussions about specific attacks, the MAC tags, and the differences between the MPC protocol SPDZ and CAPA.

This chapter is based on two works, the first one published in CRYPTO 2018 [158], and the second one published in Transactions on Cryptographic Hardware and Embedded Systems (TCHES) 2019 [74]. These works were carried in collaboration with several colleagues, including Begül Bilgin, Lauren De Meyer, Svetla Nikova, Ventzislav Nikov, Oscar Reparaz, Vincent Rijmen, and Nigel Smart. Apart from being present in the discussions, my contributions include: the hardware implementation of CAPA for both AES and Katan versions, and the implementation of M&M AES V2, the SCA validation of CAPA Katan and M&M AES V2, and the design and implementation of the FA verification procedure performed in M&M.

5.1 The Tile-Probe-and-Fault Model

The purpose of this section is to introduce a new adversarial model that serves as a basis for our security guarantees. This model expands on the wire-probe model and brings it closer to real-world processor designs. Our model is set in an architecture that mimics the actively secure MPC setting that inspires our countermeasures (see Figure 5.1). Instead of individual wires at the foundation of the model, we envisage a separation of the chip (integrated circuit) into areas or tiles, consisting of not only many wires, but also complete blocks of combinational and sequential logic. Such tiled designs are inherent in many

modern processor architectures, where the tiles correspond to "cores" and the wires correspond to the on-chip interconnect. This can easily be related to a standard MPC architecture where each tile behaves like a separate party. The main difference between our architecture and the MPC setting is that in the latter, parties are assumed to be connected by a complete network of authenticated channels. In our architecture, we know exactly how the wires are connected in the circuit.

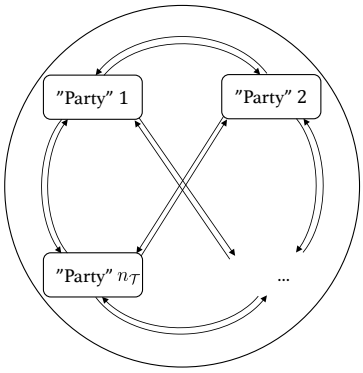


Figure 5.1: Partition of the integrated circuit area into tiles, implementing MPC "parties"

The tile architecture satisfies the independent leakage assumption [78] amongst tiles. That is, leakage is local and thus observing the behaviour of a tile by means of probing, faulting or observing its side-channel leakage, does not give unintended information about another tile through, for example, coupling [62].

As the name implies, the adversary in our model exploits side-channels and introduces faults. We stress that our goal is to *detect* faults as opposed to *tolerate* or *correct* them. That is, if an adversary interjects a fault, we want our system to abort without revealing any of the underlying secrets.

Tile Architecture. Consider a partition of the chip in a number of tiles $\mathcal{T}_i$, with wires running between each pair of tiles as shown in Figure 5.1. We call the set of all tiles $\mathcal{T}$. Each tile $\mathcal{T}_i \in \mathcal{T}$ possesses its own combinational logic, control logic (or program code) and pseudo-random number generator needed for the calculations of one share. In the abstract setting, we consider each tile as the set composed of all input and intermediate values on the wires and memory elements of those blocks. A probe-and-fault attacker may obtain, for a given subset of tiles, *all* the internal information at given time intervals on this set of tiles. He may also inject faults (known or random) into each tile in this set.

In our model, each sensitive variable is split into $n_{\mathcal{T}}$ shares through secret sharing. Without loss of generality, we use Boolean sharing in this paper.

We define each tile such that it stores and manipulates at most one share of each intermediate variable. Any wire running from one tile to another carries only blinded versions of a sensitive variables' share used by $\mathcal{T}_i$. We make minimal assumptions on the security of these wires. Instead, we include all the information on the unidirectional wires in Figure 5.1 in the tile on the *receiving* and not the *sending* end. We thus assume only one tile is affected by an integrity failure of a wire. We assume that shared calculations are performed in parallel without loss of generality. The redundancy of intermediate variables and logic makes the tiles completely independent from the communication through wires.

Probes. Throughout this work, we assume a powerful n_p-probing adversary where we give an attacker information about all intermediate values possessed by n_p specified tiles, *i.e.* $\cup_{i \in i_1,\ldots,i_{n_p}} \mathcal{T}_i$. The attacker obtains *all* the intermediate values on the tile (such as internal wire and register values) with probability one and obtains these values from the start of the computation until the end. Note that this is stronger than both the standard t-probing adversary which gives access to only t intermediate *values* within a certain amount of time [108] and ε-probing adversary where the information about t intermediate *values* is gained with certain probability. In our n_p-probing model, the adversary gets information from n intermediate values from n_p tiles where $n \gg n_p$. Therefore, our n_p-probing model is more generic and covers realistic scenarios including an attacker with a limited number of EM probes which enable observation of multiple intermediate values simultaneously within arbitrarily close proximity on the chip.

Faults. We also consider two types of fault models. Firstly, a n_f-faulting adversary which can induce chosen-value faults in any number of intermediate bits/values within n_f tiles, *i.e.* from the set $\cup_{i \in i_1,\ldots,i_{n_f}} \mathcal{T}_i$. These faults can have the nature of *either* flipping the intermediate values with a pre-calculated (adversarially chosen) offset *or* setting the intermediate values to a chosen fixed value. In particular, the faults are not limited in hamming weight. One can relate this type of faults with, for example, very accurate laser injections.

Secondly, we consider an ϵ-faulting adversary which is able to insert a random-value fault in *any* variable belonging to *any* party. This is a somehow new MPC model, and essentially means that all parties are randomly corrupted. The ϵ-adversary may inject the random-value fault according to some distribution (for example, flip each bit with certain probability), but he cannot set all intermediates to a chosen fixed value. This adversary is different from the

n_f-faulting adversary. One can relate the ϵ-faulting adversary to a certain class of non-localised EM attacks.

Time periods. We assume a notion of time periods; where the period length is at least one clock cycle. We require that an n_f-fault to an adversarially chosen value cannot be preceded by a probe within the same time period. Thus adversarial faults can only depend on values from previous time periods. This time restriction is justified by practical experimental constraints; where the time period is naturally upper bounded by the time it takes to set up such a specific laser injection.

Adversarial Models. Given the aforementioned definitions, we consider on the one hand an active adversary $\mathscr{A}_1$ with both n_p-probing and n_f-faulting capabilities simultaneously. We define $\mathcal{P}_1$ the set of up to n_p tiles that can be probed and $\mathcal{F}_1$ the set of up to n_f tiles that can be faulted by $\mathscr{A}_1$. Since each tile potentially sees a different share of a variable and we use a $n_{\mathcal{T}}$-sharing for each variable, we constrain the attack surface (the sets of adversarially probed and potentially modified tiles) as follows:

$$(\mathcal{F}_1 \cup \mathcal{P}_1) \subseteq \cup_{j=1}^{n_{\mathcal{T}}-1} \mathcal{T}_{i_j}$$

The constraint implies that at least one share remains unaccessed/honest and thus $|\mathcal{F}_1 \cup \mathcal{P}_1| \leq n_{\mathcal{T}} - 1$. Within those $n_{\mathcal{T}} - 1$ tiles, the adversary can probe and fault arbitrarily many wires, including the wires *arriving* at each tile. The adversary's n_f-faulting capabilities are limited in time by our definition of time periods, which implies that any n_f-fault cannot be preceded by another probe within the same time period.

We also consider an active adversary $\mathscr{A}_2$ that has n_p-probing and ϵ-faulting capabilities simultaneously. In this case, the constraint on the set of probed tiles $\mathcal{P}_2$ remains the same:

$$\mathcal{P}_2 \subseteq \cup_{j=1}^{n_{\mathcal{T}}-1} \mathcal{T}_{i_j}$$

but the set of faulted tiles is no longer constrained:

$$\mathcal{F}_2 \subseteq \mathcal{T}$$

Moreover, as ϵ-faults do not require the same set-up time as n_f-faults, they are not limited in time. Note that, ϵ-faults do not correspond to a standard adversary model in the MPC literature; thus this part of our model is a unique aspect of our side-channel and fault analysis focus. A rough equivalent model in the MPC literature would be for an honest-but-curious adversary who is

able to replace the share or MAC values of honest players with values selected from a given random distribution. Whilst such an attack makes sense in the hardware model we consider, in the traditional MPC literature this model is of no interest due to the supposed isolated nature of computing parties.

As our constructions are based on MPC protocols which are *statically secure* we make the same assumptions in our tile-probe-and-fault model, *i.e.* the selection of tiles attacked must be fixed beforehand and cannot depend on information gathered during computation. This model reflects realistic attackers since it is infeasible to move a probe or a laser during a computation with today's resources. We thus assume that both adversaries $\mathscr{A}_1$ and $\mathscr{A}_2$ are static.

On tiles and shares

In the masking literature, the typical convention to designate the degree of side-channel security is to use d. This means that sharings are typically referred to as $d+1$ (or $td+1$) sharing to provide d^{th}-order SCA security. For this chapter, we additionally introduce $n_{\mathcal{T}}$ to refer to the number of tiles in the Tile-Probe-and-Fault model. On the one hand, in CAPA, a design with $n_{\mathcal{T}}$ number of tiles provides $(n_{\mathcal{T}}-1)^{th}$-order SCA security. On the other hand, in M&M, since we employ a $d+1$ masking scheme, we utilize the same convention as in the masking community, where our scheme is instantiated with $d+1$ shares to provide d^{th}-order SCA security. Thus, in M&M $n_{\mathcal{T}} = d+1$.

5.2 CAPA

We introduce CAPA, a countermeasure against the tile-probe-and-fault-attacker. CAPA inherits theoretical aspects of the MPC protocol SPDZ [72] by similarly computing on shared values, along with corresponding shared MAC tags. The former prevents the adversary from learning sensitive values, while the latter allows for detection of any faults introduced. Moreover, having originated from the MPC protocol SPDZ, CAPA is the first countermeasure with *provable* security against *combined* attacks. The methodology can be scaled to achieve an arbitrary fault detection probability and is suitable for implementation in both hardware and software.

The CAPA methodology consists of two stages. A preprocessing step generates auxiliary data, which is used to perform the actual cryptographic operation in the evaluation step. We first present some notation, then the building blocks for the main evaluation, and finally the preprocessing components.

Notation. Although generalization to any finite field holds, we work over a field $\mathbb{F}_q$ with characteristic 2, for example $GF(2^k)$ for a given k, as this is sufficient for application to most symmetric ciphers. We use $\cdot$ and $+$ to describe multiplication and addition in $\mathbb{F}_q$ respectively. We use upper case letters for constants. The lower case letters x, y, z are reserved for the variables used only in the evaluation stage (e.g. sensitive variables) whereas $a, b, c, \ldots$ represent auxiliary variables generated from randomness in the preprocessing stage. The kronecker delta function is denoted by $\delta_{i,j}$. We use $L(.)$ to denote a linear function, and $A(.) = C + L(.)$ to denote an additively homomorphic function with C some constant.

Information Theoretic MAC Tags and the MAC Key α. We represent a value $a \in \mathbb{F}_q$ (similarly $x \in \mathbb{F}_q$) as a pair $\langle \vec{a} \rangle = (\vec{a}, \vec{\tau}^a)$ of data and tag shares in the masked domain. The data shares $\vec{a} = (a_1, \ldots, a_{n_\mathcal{T}})$ satisfy $\sum a_i = a$. For each $a \in \mathbb{F}_q$, there exists a corresponding MAC tag τ^a computed as $\tau^a = \alpha \cdot a$, where α is a MAC key, which is secret-shared amongst the tiles as $\alpha = \sum \alpha_i$, and must be fresh for every encryption.

Analogously to the data, the MAC tag is shared $\vec{\tau}^a = (\tau_1^a, \ldots, \tau_{n_\mathcal{T}}^a)$, such that it satisfies $\sum \tau_i^a = \tau^a$, but the MAC key itself does not carry a tag. Depending on a security parameter m, there can be m independent MAC keys $\alpha[j] \in \mathbb{F}_q$ for $j \in \{1, \ldots, m\}$. In that case, α as well as τ^a are in $\mathbb{F}_q^m$ and the tag shares satisfy $\sum \tau_i^a[j] = \tau^a[j] = \alpha[j] \cdot a$, $\forall j \in \{1, \ldots, m\}$. This is crucial in case the field $\mathrm{GF}(2^k)$ is too small. Further we assume $m = 1$ unless otherwise mentioned.

Note that the MAC key α authenticates the sensitive value a itself and not its shares a_i. Hence, the tag shares τ_i^a are not tags of the value shares a_i but rather a share of the tag τ^a:

$$\tau_i^a \neq \alpha \cdot a_i$$

$$\sum_i \tau_i^a = \alpha \cdot \sum_i a_i$$

5.2.1 Evaluation Stage

We let each tile $\mathcal{T}_i$ hold the ith share of each sensitive and auxilary variable $(x_i, \ldots, a_i, \ldots)$ and the MAC key share α_i. We first describe operations that do not require communication between tiles.

Addition. To compute the addition $(\vec{z}, \overrightarrow{\tau^z})$ of $(\vec{x}, \overrightarrow{\tau^x})$ and $(\vec{y}, \overrightarrow{\tau^y})$, each tile performs local addition of their data shares $z_i = x_i + y_i$ and their tag shares

$\tau_i^z = \tau_i^x + \tau_i^y$. When one operand is public (for example, a cipher constant $C \in \mathbb{F}_q$), the sum can be computed locally as $z_i = x_i + C \cdot \delta_{i,1}$ for value shares and $\tau_i^z = \tau_i^x + C \cdot \alpha_i$ for tag shares.

Multiplication by a Public Constant. Given a public constant $C \in \mathbb{F}_q$, the multiplication $(\vec{z}, \overrightarrow{\tau^z})$ of $(\vec{x}, \overrightarrow{\tau^x})$ and C is obtained locally by setting $z_i = C \cdot x_i$ and $\tau_i^z = C \cdot \tau_i^x$.

The following operations, on the other hand, require auxiliary data generated in a preprocessing stage and also communication between the tiles.

Multiplication. Multiplication of $(\vec{x}, \overrightarrow{\tau^x})$ and $(\vec{y}, \overrightarrow{\tau^y})$ requires as auxiliary data a Beaver *triple* $(\langle\vec{a}\rangle, \langle\vec{b}\rangle, \langle\vec{c}\rangle)$ [24], which satisfies $c = a \cdot b$, for random a and b. The multiplication itself is performed in four steps.

- Step A. In the *blinding* step, each tile $\mathcal{T}_i$ computes locally a randomized version of its share of the secret: $\varepsilon_i = x_i + a_i$ and $\eta_i = y_i + b_i$.
- Step B. In the *partial unmasking* step, each tile $\mathcal{T}_i$ broadcasts its own shares ε_i and η_i to other tiles, such that each tile can construct and store locally the values $\varepsilon = \sum \varepsilon_i$ and $\eta = \sum \eta_i$. The value ε (resp. η) is the *partial unmasking* of $(\vec{\varepsilon}, \overrightarrow{\tau^\varepsilon})$ (resp. $(\vec{\eta}, \overrightarrow{\tau^\eta})$), *i.e.* the value ε (resp. η) is unmasked but its tag $\overrightarrow{\tau^\varepsilon}$ ($\overrightarrow{\tau^\eta}$) remains shared. These values are blinded versions of the secrets x and y and can therefore be made public.
- Step C. In the *MAC-tag checking* step, the tiles check whether the tags $\overrightarrow{\tau^\varepsilon}$ ($\overrightarrow{\tau^\eta}$) are consistent with the public values ε and η, using a method which we will explain later in this section.
- Step D. In the *Beaver computation* step, each tile locally computes

$$z_i = c_i + \varepsilon \cdot b_i + \eta \cdot a_i + \varepsilon \cdot \eta \cdot \delta_{i,1}$$

$$\tau_i^z = \tau_i^c + \varepsilon \cdot \tau_i^b + \eta \cdot \tau_i^a + \varepsilon \cdot \eta \cdot \alpha_i.$$

It can be seen easily that the sharing $(\vec{z}, \overrightarrow{\tau^z})$ corresponds to $z = x \cdot y$ unless faults occurred. Step B and C are the only steps that require communication among tiles. Step A and D are completely local. Note that to avoid leaking information on the sensitive data x and y, the shares ε_i and η_i must be synchronized using memory elements after step A, before being released to other tiles in step B. Thus, non-completeness is preserved in every stage and unexpected dependencies

coming from glitches are prevented. Moreover, we remark that step C does not require the result of step B and can thus be performed in parallel.

Squaring. Squaring is a linear operation in characteristic 2 fields. Hence, the output shares of a squaring operation can be computed locally using the input shares. However, obtaining the corresponding tag shares is non-trivial sincen we cannot just square the tags, otherwise we would get $(\tau^x)^2 = \alpha^2 \cdot x^2 \neq \tau^{x^2} = \alpha \cdot x^2$. To square $(\vec{x}, \overrightarrow{\tau^x})$ into $(\vec{z}, \overrightarrow{\tau^z})$, we therefore require an auxiliary *tuple* $(\langle\vec{a}\rangle, \langle\vec{b}\rangle)$ such that $b = a^2$. The procedure to obtain $(\vec{z}, \overrightarrow{\tau^z})$ mimics that of multiplication with some modifications: there is only one partially unmasked value $\varepsilon = x + a$, whose tag needs to be checked, and each tile calculates $z_i = b_i + \varepsilon^2 \cdot \delta_{i,1}$ and $\tau_i^z = \tau_i^b + \varepsilon^2 \cdot \alpha_i$.

Following the same spirit, we can also perform the following operations.

Affine Transformation. Provided that we have access to a tuple $(\langle\vec{a}\rangle, \langle\vec{b}\rangle)$ such that $b = A(a)$, we can compute $(\vec{z}, \overrightarrow{\tau^z})$ satisfying $z = A(x) = C + L(x)$, where $L(x)$ is an additively homomorphic function over the finite field, by computing the output sharing as $z_i = b_i + L(\varepsilon) \cdot \delta_{i,1}$ and $\tau_i^z = \tau_i^b + L(\varepsilon) \cdot \alpha_i$.

Multiplication following Linear Transformations. The technique used for the above additively homomorphic operations can be generalized even further to compute $z = L_1(x) \cdot L_2(y)$ in shared form, where L_1 and L_2 are additively homomorphic functions. A trivial methodology would require two tuples $(\langle\overrightarrow{a_i}\rangle, \langle\overrightarrow{b_i}\rangle)$ with $b_i = L_i(a_i)$ for $i \in \{1, 2\}$, plus a standard Beaver triple (*i.e.* requiring seven pre-processed data items). We see that we can do the same operation with five pre-processed items $(\langle\vec{a}\rangle, \langle\vec{b}\rangle, \langle\vec{c}\rangle, \langle\vec{d}\rangle, \langle\vec{e}\rangle)$, such that $c = L_1(a)$, $d = L_2(b)$ and $e = L_1(a) \cdot L_2(b)$. The tiles partially unmask $\vec{x} + \vec{a}$ (resp. $\vec{y} + \vec{b}$) to obtain ε (resp. η) and verify them. Each tile computes its value share and tag share of z as $z_i = e_i + L_1(\varepsilon) \cdot d_i + L_2(\eta) \cdot c_i + L_1(\varepsilon) \cdot L_2(\eta) \cdot \delta_{i,1}$ and $\tau_i^z = \tau_i^e + L_1(\varepsilon) \cdot \tau_i^d + L_2(\eta) \cdot \tau_i^c + L_1(\varepsilon) \cdot L_2(\eta) \cdot \alpha_i$, respectively. We refer to $(\langle\vec{a}\rangle, \langle\vec{b}\rangle, \langle\vec{c}\rangle, \langle\vec{d}\rangle, \langle\vec{e}\rangle)$ as a *quintuple*.

Proof.

$$\sum_{i=1}^{n_\mathcal{T}} z_i = \sum_{i=1}^{n_\mathcal{T}} \left(e_i + L_1(\varepsilon) \cdot d_i + L_2(\eta) \cdot c_i \right) + L_1(\varepsilon) \cdot L_2(\eta)$$

$$= \sum_{i=1}^{n_{\mathcal{T}}} e_i + L_1(\varepsilon) \cdot \sum_{i=1}^{n_{\mathcal{T}}} d_i + L_2(\eta) \cdot \sum_{i=1}^{n_{\mathcal{T}}} c_i + L_1(\varepsilon) \cdot L_2(\eta)$$

$$= L_1(a) \cdot L_2(b) + L_1(x+a) \cdot L_2(b) + L_2(y+b) \cdot L_1(a) + L_1(x+a) \cdot L_2(y+b)$$

$$= L_1(a) \cdot L_2(b) + L_1(x) \cdot L_2(b) + L_1(a) \cdot L_2(b) + L_1(a) \cdot L_2(y) + L_1(a) \cdot L_2(b)$$

$$+ L_1(x) \cdot L_2(y) + L_1(x) \cdot L_2(b) + L_1(a) \cdot L_2(y) + L_1(a) \cdot L_2(b)$$

$$= L_1(x) \cdot L_2(y)$$

$$\sum_{i=1}^{n_{\mathcal{T}}} \tau_i^z = \sum_{i=1}^{n_{\mathcal{T}}} \left(\tau_i^e + L_1(\varepsilon) \cdot \tau_i^d + L_2(\eta) \cdot \tau_i^c + L_1(\varepsilon) \cdot L_2(\eta) \cdot \alpha_i \right)$$

$$= \sum_{i=1}^{n_{\mathcal{T}}} \tau_i^e + L_1(\varepsilon) \cdot \sum_{i=1}^{n_{\mathcal{T}}} \tau_i^d + L_2(\eta) \cdot \sum_{i=1}^{n_{\mathcal{T}}} \tau_i^c + L_1(\varepsilon) \cdot L_2(\eta) \cdot \sum_{i=1}^{n_{\mathcal{T}}} \alpha_i$$

$$= \alpha \cdot e + L_1(\varepsilon) \cdot \alpha \cdot d + L_2(\eta) \cdot \alpha \cdot c + L_1(\varepsilon) \cdot L_2(\eta) \cdot \alpha$$

$$= \alpha \cdot \left(e + L_1(\varepsilon) \cdot d + L_2(\eta \cdot c + L_1(\varepsilon) \cdot L_2(\eta) \right)$$

$$= \alpha \cdot L_1(x) \cdot L_2(y)$$

□

Checking the MAC Tag of Partially Unmasked Values.

Consider a public value $\varepsilon = x + a$, calculated in the partial unmasking step of the Beaver multiplication operation. Recall that we obtain its MAC-tag shares as follows: $\tau_i^\varepsilon = \tau_i^a + \tau_i^x$. During the MAC-tag checking step of the Beaver operation, the authenticity of τ^ε corresponding to ε is tested. As ε is public, each tile can calculate and broadcast the value $\varepsilon \cdot \alpha_i + \tau_i^\varepsilon$. For a correct tag, we expect $\sum \tau_i^\varepsilon = \alpha \cdot \varepsilon$, thus each tile computes $\sum(\varepsilon \cdot \alpha_i + \tau_i^\varepsilon)$ and proceeds if the result is zero. Recall that the broadcasting must be preceded by a synchronization of the shares.

Note on Unmasked Values/Calculations.

There are several components in a cipher which do not need to be protected against SCA (*i.e.* masked), because their specific values are not sensitive. One prominent example is the control unit which decides what operations should be

performed (*e.g.* the round counter). Other examples are constants such as the AES affine constant `0x63` or public values such as ε in a Beaver calculation and the difference $\varepsilon \cdot \alpha + \tau^{\varepsilon}$ during the MAC-tag checking phase.

While these *public* components are not sensitive in a SCA context, they *can* be targeted in a fault attack. It is therefore important to introduce some redundancy. Each tile should have its own control logic and keep a local copy of all public values to avoid single points of attack. The shares ε_i are distributed to *all* tiles so that ε can be unmasked by each tile separately and any subsequent computation performed on these public values is repeated by each tile. Finally, each tile also keeps its own copy of the abort status. This is in fact completely analogous to the MPC scenario.

5.2.2 Preprocessing Stage

The auxiliary data $(\langle\vec{a}\rangle, \langle\vec{b}\rangle, \ldots)$ required in the Beaver evaluations, is generated in a preprocessing stage. This preparation corresponds to the *offline* phase in SPDZ . However, CAPA's preprocessing stage is lighter and does not require a public key calculation due to the differences in adversary model. As in SPDZ , this stage is completely independent from the sensitive data of the main evaluation. Below, we describe the generation of a Beaver triple used in multiplication. This can trivially be generalized to tuples and quintuples.

Auxiliary Data Generation. To generate a triple $(\langle\vec{a}\rangle, \langle\vec{b}\rangle, \langle\vec{c}\rangle)$ satisfying $c = a \cdot b$, we draw random shares $\vec{a} = (a_1, \ldots, a_{n_\mathcal{T}})$ and $\vec{b} = (b_1, \ldots, b_{n_\mathcal{T}})$ and use a passively secure shared multiplier to compute $\vec{c}$ s.t. $c = a \cdot b$. We then use another such multiplication with the shared MAC key $\vec{\alpha}$ to generate tag shares $\overrightarrow{\tau^a}$, $\overrightarrow{\tau^b}$, $\overrightarrow{\tau^c}$. We note that the shares a_i, b_i are randomly generated by tile $\mathcal{T}_i$. There are thus $n_\mathcal{T}$ separate PRNG's on $n_\mathcal{T}$ distinct tiles.

Passively Secure Shared Multiplier. For a secure implementation of a shared multiplication, no subset of $n_\mathcal{T} - 1$ tiles should have access to all shares of any variable. This concept, which is used in the context of secure implementations against SCA on hardware, is precisely called $(n_\mathcal{T} - 1)^{\text{th}}$-order non-completeness in [40, 148]. In the last decade, there has been significant improvement on passively secure shared multipliers that can be used in both hardware and software [40, 97, 98, 147, 157]. In principle, CAPA can use any such multiplier as long as the tile structure still holds.

A close inspection of existing multipliers shows that they require the calculation of the cross products $a_i b_j$. In order to make these multipliers compatible with the CAPA tile architecture, we define tiles $\mathcal{T}_{i,j}$ which receive a_i from $\mathcal{T}_i$ and b_j from $\mathcal{T}_j$ where $i \neq j$ in order to handle the pair (a_i, b_j) to be used during tuple, triple and quintuple generation. This implies $n_\mathcal{T}(n_\mathcal{T} - 1)$ smaller tiles used only during auxiliary data generation in addition to $n_\mathcal{T}$ tiles used for both auxiliary data generation and evaluation. The output wires from $\mathcal{T}_{i,j}$ are only connected to $\mathcal{T}_i$ and carry randomized information.

The multipliers used in the preprocessing phase are only *passively* secure. We also ensure resistance against *active* adversaries because on the one hand, deterministic faults are limited to $n_\mathcal{T} - 1$ tiles and on the other, because of a relation verification step, which is explained in the next section.

5.2.3 Relation Verification of Auxiliary Data

The information theoretic MAC tags provide security against faults induced in the evaluation stage. To detect faults in the preprocessing stage, we perform a relation verification of the auxiliary data. This relation verification step is done for each generated triple that is passed from the preprocessing to the evaluation stage and ensures that the triple is functionally correct (*i.e.* $c = a \cdot b$) by sacrificing another triple. That is, we take as input two triples $(\langle\vec{a}\rangle, \langle\vec{b}\rangle, \langle\vec{c}\rangle)$ and $(\langle\vec{d}\rangle, \langle\vec{e}\rangle, \langle\vec{f}\rangle)$, that should satisfy the same relation, in this example $c = a \cdot b$ and $f = d \cdot e$. The following Beaver computation holds if and only if both relations are satisfied:

1. Draw a random $r_1 \in \mathbb{F}_q$
2. Use triple $(\langle\vec{d}\rangle, \langle\vec{e}\rangle, \langle\vec{f}\rangle)$ to calculate the multiplication of $r_1 \cdot \langle\vec{a}\rangle$ and $\langle\vec{b}\rangle$ using a constant multiplication with r_1, followed by the Beaver equation for multiplication described above. The result $\langle\vec{\tilde{c}}\rangle$ is a shared representation of $\tilde{c} = r_1 \cdot a \cdot b$.
3. For each share i, calculate the difference with the shares and tags of $r_1 \cdot c$: $\Delta_i = r_1 \cdot c_i + \tilde{c}_i$ and $\tau_i^\Delta = r_1 \cdot \tau_i^c + \tau_i^{\tilde{c}}$.
4. Unmask the resulting differences Δ and τ^Δ.
5. If a difference is nonzero, reject $(\langle\vec{a}\rangle, \langle\vec{b}\rangle, \langle\vec{c}\rangle)$ as a valid triple.
6. Pick another $r_2 \in \mathbb{F}_q$ such that $r_2 \neq r_1$ and repeat a second time.

Note that this relation verification ensures that the second triple is functionally correct too. However, it is burnt (or "sacrificed") in this process in order to ensure that the first triple can be used securely further on. Note that this relation verification or "sacrificing" step is mandatory in each Beaver-like operation.

Why We Need Randomization. This sacrificing step involves two values r_1 and r_2. We present the following attack to illustrate why this randomization is needed. Again, we elaborate on triples, but the same can be said for tuples and quintuples. As the security does not rely on the secrecy of r_1 and r_2, we assume for simplicity that they are known to the attacker. We only stress that they are different: $r_1 \neq r_2$.

Consider two triples $(\langle\vec{a}\rangle, \langle\vec{b}\rangle, \langle\vec{c'}\rangle)$ and $(\langle\vec{d}\rangle, \langle\vec{e}\rangle, \langle\vec{f'}\rangle)$ at the input of the sacrificing stage. We assume that the adversary has introduced an additive difference into one share of c' and f' such that $c' = a \cdot b + \Delta_c$ and $f' = d \cdot e + \Delta_f$. This fault is injected before the MAC tag calculation, so that $\tau^{c'}$ and $\tau^{f'}$ are valid tags for the faulted values c' and f' respectively. In particular, this means we have $\tau^{c'} = \tau^c + \alpha \cdot \Delta_c$ and $\tau^{f'} = \tau^f + \alpha \cdot \Delta_f$.

The sacrificing step calculates the following four differences (for $r_j = r_1$ and r_2) and only succeeds if all are zero.

$$\begin{aligned}
\Delta_j &= \sum_{i=1}^{n_{\mathcal{T}}} \left(r_j \cdot c'_i + f'_i + \varepsilon \cdot e_i + \eta \cdot d_i \right) + \varepsilon \cdot \eta \\
&= r_j \cdot \Delta_c + \Delta_f \stackrel{?}{=} 0 \\
\tau^{\Delta_j} &= \sum_{i=1}^{n_{\mathcal{T}}} \left(r_j \cdot \tau_i^{c'} + \tau_i^{f'} + \varepsilon \cdot \tau_i^e + \eta \cdot \tau_i^d + \varepsilon \cdot \eta \cdot \alpha_i \right) \\
&= r_j \cdot \alpha \cdot \Delta_c + \alpha \cdot \Delta_f \stackrel{?}{=} 0
\end{aligned}$$

Without randomization (*i.e.* $r_1 = r_2 = 1$), the attacker only has to match the differences $\Delta_f = \Delta_c$ to pass verification. With a random r_1, the attacker can fix $\Delta_f = r_1 \cdot \Delta_c$ to automatically force Δ_1 and τ^{Δ_1} to zero. Even if he does not know r_1, he has probability as high as 2^{-k} to guess it correctly.

Only thanks to the repetition of the relation verification with r_2, the adversary is detected with a probability $1 - 2^{-km}$. Assuming he fixed $\Delta_f = r_1 \cdot \Delta_c$, it is impossible to also achieve $\Delta_f = r_2 \cdot \Delta_c$. Even if the attacker manages to force Δ_2 to zero with an additive injection (since he knows all components r_2, Δ_c and Δ_f), he cannot get rid of the difference $\tau^{\Delta_2} = r_2 \cdot \alpha \cdot \Delta_c + \alpha \cdot \Delta_f$ without

knowing the MAC key. Since α remains secret, the attacker only has a success probability of 2^{-km} to succeed.

5.2.4 Hardware Implementations

We now describe two case studies for applying CAPA in hardware. Our implementations are somewhat optimized for latency rather than area with $n_{\mathcal{T}}$ tiles spatially separated and operating in parallel, each with its own combinational and control logic and auxiliary data preparation module. These preparation modules are equipped with a passively secure shared multiplication with higher-order non-completeness. In order to minimize the randomness requirement, our implementation uses the multiplier from [98], hereafter referred to as DOM.

Case Study: KATAN-32

KATAN-32 is a shift register based block cipher, which has an 80-bits key and processes 32-bit plaintext input. It is designed specifically for efficient hardware implementations and performs 254 cycles of four AND-XOR operations. Hence, its natural shared data representation is in the field $\mathbb{F}_q = \mathrm{GF}(2)$, which makes the mapping into CAPA operations relatively straightforward. However, the small finite field means that we need to utilize a vectorized MAC-tag operation ($m > 1$) to ensure a good probability of detecting errors. Our implementation is round based, as in [56] with three AND-XOR Beaver operations and one constant AND-XOR calculated in parallel. Each Beaver AND-XOR operation requires two cycles, and is implemented in a pipelined fashion such that the latency of the whole computation increases only by one clock cycle.

Implementation Cost. Tables 5.1 and 5.2 summarize the area of our KATAN implementations. Naturally, compared to a shared implementation without MAC tags, the state registers grow with a factor $m + 1$ as the MAC-key size increases. In the last columns, we extrapolate the area results for any m.

Each Beaver multiplication in GF(2) requires one triple, and each triple needs $2n_{\mathcal{T}}$ random bits for generating $\vec{a}$ and $\vec{b}$. A $n_{\mathcal{T}}$-share DOM multiplication requires $\binom{n_{\mathcal{T}}}{2}$ units of randomness. The construction of one triple requires $1 + 3m$ masked multiplications: one to obtain the multiplication $\vec{c}$ of $\vec{a}$ and $\vec{b}$; and $3m$ to obtain the m tags $\vec{\tau}^a$,$\vec{\tau}^b$ and $\vec{\tau}^c$. Due to the relation verification through the sacrificing of another triple, the randomness must be doubled.

Table 5.1: Area (GE) of 2-share KATAN-32 implementations with m MAC keys $\alpha[j] \in \mathbb{F}_q$

	No tags	$m = 1$	$m = 8$	Any m
- Evaluation	2 315	4 708	21 404	$\approx 2\,315 + 2\,390m$
* Shift Register	888	1 823	8 419	$\approx 888 + 935m$
* Key Schedule	1 427	2 885	12 985	$\approx 1\,427 + 1\,455m$
- Preprocessing (x3)	363	679	2 727	$\approx 363 + 315m$
* Two triple generation	237	431	1 786	$\approx 237 + 195m$
* Relation verification	126	248	941	$\approx 126 + 120m$
Total	3 672	7 103	30 596	$\approx 3\,672 + 3\,430m$

Hence, the total required number of random bits per round of KATAN is $3 \cdot 2 \cdot (2n_{\mathcal{T}} + (1 + 3m)\frac{n_{\mathcal{T}}(n_{\mathcal{T}}-1)}{2}))$.

Experimental Validation. The goal of the prior proof-of-concept implementation is to experimentally validate the protection against side-channel attacks offered by the CAPA methodology. We evaluate first- and second-order secure KATAN instances using the same set-up and methodology as detailed in § 2.8. The KATAN implementations use $n_{\mathcal{T}} = 2$ (resp. $n_{\mathcal{T}} = 3$) shares and $m = 2$ MAC keys. The parameter $m = 2$ is insufficient in practice, but serves for this experiment since m has no influence on SCA security. The designs are clocked at 3 MHz and we sample power traces of 10 000 time samples each at 1 GS/s. Exemplary traces are shown in Figure 5.2, top.

In Figure 5.2, we show the results for the first-order secure design ($n_{\mathcal{T}} = 2$). In the left column, the PRNG is turned off, emulating an unmasked design. Indeed, we see clear leakage at first order, since the t-statistics cross the threshold 4.5. With the PRNG on (right column), no first-order leakage is detected with up to 100 million traces. As expected, we do see second-order leakage.

Figure 5.3 exhibits the results for the second-order secure design ($n_{\mathcal{T}} = 3$). The

Table 5.2: Area (GE) of 3-share KATAN-32 implementations with m MAC keys $\alpha[j] \in \mathbb{F}_q$

	No tags	$m = 1$	$m = 8$	Any m
- Evaluation	3 560	7 139	32 368	$\approx 3\,560 + 3\,580m$
* Shift Register	1 363	2 812	12 890	$\approx 1\,363 + 1\,450m$
* Key Schedule	2 197	4 327	19 478	$\approx 2\,197 + 2\,130m$
- Preprocessing (x3)	638	1 468	7 124	$\approx 638 + 830m$
* Two triple generation	428	952	4 694	$\approx 428 + 524m$
* Relation verification	210	516	2 430	$\approx 210 + 306m$
Total	5 971	12 083	55 254	$\approx 5\,971 + 6\,112m$

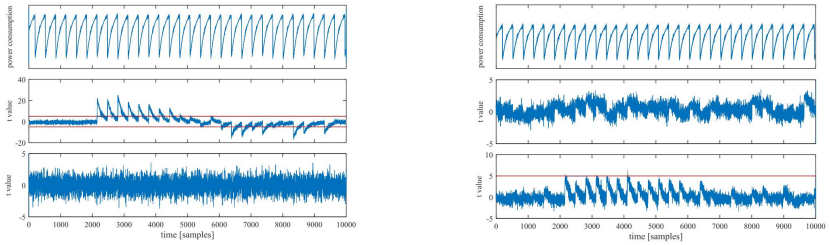


Figure 5.2: Non-specific leakage detection on the first 31 rounds of first-order KATAN. Left column: PRNG off (24K traces). Right column: PRNG on (100M traces). Rows (top to bottom): exemplary power trace; first-order t-test; second-order t-test

left column shows clear leakage at first-, and third- order when the PRNG is turned off. In the right column, we repeat the procedure with PRNG on and no univariate leakage is detected with up to 100 million traces.[1]

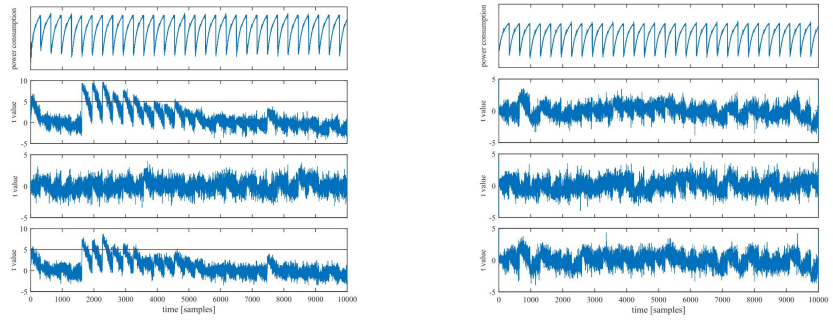


Figure 5.3: Non-specific leakage detection on the first 31 rounds of second-order KATAN. Left column: PRNG off (24K traces). Right column: PRNG on (100M traces). Rows (top to bottom): exemplary power trace; first-order t-test; second-order t-test; third-order t-test

Fault Analysis validation. By the time we designed this architecture, no automated procedures to evaluate the security against FA were available. We rely on the strong theory behind our scheme, and claim FA resistance in accordance to the theoretical bounds. Later in Chapter 6, we devise an automated procedure

[1]Since our implementation handles 3 shares, we expect to detect leakage in the third order. Due to platform noise, this is not visible.

to evaluate the performance of a FA countermeasure, and present a thorough evaluation of a CAPA protected module.

Case Study: AES

There has been a great deal of work on MPC and masked implementations of the basic AES operations. We take what has now become the traditional approach and work in the field GF(2^8) with $m = 1$ for AES, *i.e.* the MAC key, data and tag shares α_i, a_i and τ_i^a are $\in$ GF(2^8). The ShiftRows and MixColumns operations are linear in GF(2^8), and hence, straightforward. Here, we only describe the S-Box calculation.

Design choices. The AES S-Box consists of an inversion in GF(2^8), followed by an affine transformation over bits. We distinguish two methodologies for the S-Box implementation: It is well known that the combination of the two operations can be expressed by the following polynomial in GF(2^8) [71]:

$$\begin{aligned}\text{S-Box}(x) =& \texttt{0x63} + \texttt{0x8F} \cdot x^{127} + \texttt{0xB5} \cdot x^{191} + \texttt{0x01} \cdot x^{223} + \texttt{0xF4} \cdot x^{239} \\ &+ \texttt{0x25} \cdot x^{247} + \texttt{0xF9} \cdot x^{251} + \texttt{0x09} \cdot x^{253} + \texttt{0x05} \cdot x^{254}\end{aligned} \tag{5.1}$$

This polynomial can be implemented using 6 squares and 7 multiplications in GF(2^8) with a latency of 13 clock cyles. A second approach is to evaluate the inversion $x \longrightarrow x^{254}$ using the following multiplication chain from [101]:

$$x^{254} = x^4 \cdot \left(\left((x^5)^5\right)^5\right)^2$$

For the AES affine transform $A(x)$ over GF(2), we can use the Beaver operation described in §5.2.1 to evaluate it in one cycle, using auxiliary affine tuples $(\langle\vec{a}\rangle, \langle\vec{b}\rangle)$ such that $b = A(a)$. Initial estimations reveal the former method is more expensive than the latter, so we adopt the latter technique.

Multiplication Chain. Our implementation of the proposed multiplication chain uses two types of operations: x^5 and $x^4 \cdot y^2$, which can both be computed as described in §5.2.1 (Multiplication following Linear Transformations). Given an input $\langle\vec{x}\rangle$ and a triple $(\langle\vec{a}\rangle, \langle\vec{b}\rangle, \langle\vec{c}\rangle)$ such that $b = a^4$ and $c = a^5$, we calculate the CAPA exponentiation to the power five. Likewise, we perform the map $x^4 \cdot y^2$ (with $y = x^{125}$) in one cycle, using quintuples $(\langle\vec{a}\rangle, \langle\vec{b}\rangle, \langle\vec{c}\rangle, \langle\vec{d}\rangle, \langle\vec{e}\rangle)$ such that $c = a^4$, $n_{\mathcal{T}} = b^2$ and $e = c \cdot d = a^4 \cdot b^2$. As a result, an inversion in GF(2^8) costs only 4 cycles, using 3 exponentiation triples and 1 quintuple. Combined

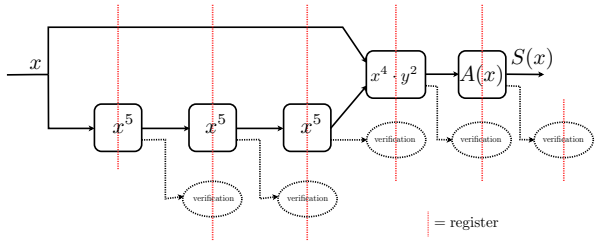


Figure 5.4: AES S-Box pipeline

with the affine stage, we obtain the S-Box output in 5 cycles (see Figure 5.4). This approach does not only optimize the number of cycles but also the amount of required randomness. The S-Box is implemented as a five stage pipeline.

Implementation Cost. We use a serialized AES architecture, based on that in [98]. One round of the cipher requires 21 clock cycles, making the latency of one complete encryption 226 clock cycles. Since the unprotected serialised implementation of [141] also requires 226 cycles, the timing performance is very good.

Table 5.3 presents the area for the different blocks that make up our AES implementation. We can see a significant difference between the preprocessing and evaluation stages, *i. e.* the efficient calculation phase comes at the cost of expensive resource generation machinery.

Table 5.4 summarizes the required number of random bytes for the generation of the triples/tuples for the AES S-Box as a function of the number of MAC keys m

Table 5.3: Areas for first- and second-order AES implementations with $m = 1$ in 2-NAND Gate Equivalents (GE)

Evaluation	$n_{\mathcal{T}} = 2$	$n_{\mathcal{T}} = 3$	Preprocessing	$n_{\mathcal{T}} = 2$	$n_{\mathcal{T}} = 3$
S-Box	18 810	28 234	Quintuples	29 147	53 212
* Beaver x^5 (x3)	3 914	5 875	* Generation	15 092	32 241
* Beaver x^4y^2	4 944	7 427	* Sacrificing	14 055	20 971
* Beaver Affine	1 563	2 344	Triples (x3)	19 106	34 954
State array	4 962	7 466	* Generation	9 804	21 112
* MixColumns	1 056	1 584	* Sacrificing	9 302	13 842
Key array	3 225	4 835	Affine tuples	7 603	14 657
Others	1 296	1 839	* Generation	4 821	10 444
			* Sacrificing	2 782	4 213
Total	28 293	42 374	Total	94 068	172 731
TOTAL				122 361	215 105

Table 5.4: The number of randomness in bytes for the initial sharing, shared multiplication and the sacrifice required for AES S-Box

	Initial sharing	Shared mult.	Total
Exp. triple	$n_{\mathcal{T}}$	$1+3m$	$2(n_{\mathcal{T}} + (1+3m)\frac{n_{\mathcal{T}}(n_{\mathcal{T}}-1)}{2})$
Quintuple	$2n_{\mathcal{T}}$	$1+5m$	$2(2n_{\mathcal{T}} + (1+5m)\frac{n_{\mathcal{T}}(n_{\mathcal{T}}-1)}{2})$
Affine tuple	$n_{\mathcal{T}}$	$2m$	$2(n_{\mathcal{T}} + 2m\frac{n_{\mathcal{T}}(n_{\mathcal{T}}-1)}{2})$
Total			$12n_{\mathcal{T}} + 2(4+16m)\frac{n_{\mathcal{T}}(n_{\mathcal{T}}-1)}{2}$

Table 5.5: Overview of the number of $\mathbb{F}_q$ multiplications (.), $\mathbb{F}_q$ additions (+) and linear operations in $GF(2)$ ($L(.)$) required to calculate all building blocks with $n_{\mathcal{T}}$ shares and m tags

	Public Values			Output calculation				MAC check	
				Value		Tags			
	·	+	$L(.)$	·	+	·	+	·	+
Add.					$n_{\mathcal{T}}$		$n_{\mathcal{T}}m$		
Add. with C					1	$n_{\mathcal{T}}m$	$n_{\mathcal{T}}m$		
Multip. with C				$n_{\mathcal{T}}$		$n_{\mathcal{T}}m$			
Multip.	$n_{\mathcal{T}}$	$2n_{\mathcal{T}} + 2(n_{\mathcal{T}}-1)n_{\mathcal{T}}$		$2n_{\mathcal{T}}$	$2n_{\mathcal{T}}+1$	$3n_{\mathcal{T}}m$	$3n_{\mathcal{T}}m$	$2n_{\mathcal{T}}m$	$4n_{\mathcal{T}}m + 2(n_{\mathcal{T}}-1)n_{\mathcal{T}}m$
Square/Affine		$n_{\mathcal{T}} + (n_{\mathcal{T}}-1)n_{\mathcal{T}}$	$n_{\mathcal{T}}$		1	$n_{\mathcal{T}}m$	$n_{\mathcal{T}}m$	$n_{\mathcal{T}}m$	$2n_{\mathcal{T}}m + (n_{\mathcal{T}}-1)n_{\mathcal{T}}m$
$L_1(x) \cdot L_2(y)$	$n_{\mathcal{T}}$	$2n_{\mathcal{T}} + 2(n_{\mathcal{T}}-1)n_{\mathcal{T}}$	$n_{\mathcal{T}}+n_{\mathcal{T}}$	$2n_{\mathcal{T}}$	$2n_{\mathcal{T}}+1$	$3n_{\mathcal{T}}m$	$3n_{\mathcal{T}}m$	$2n_{\mathcal{T}}m$	$4n_{\mathcal{T}}m + 2(n_{\mathcal{T}}-1)n_{\mathcal{T}}m$

and the number of shares $n_{\mathcal{T}}$. Recall that the S-Box needs three exponentiation triples, one quintuple and one affine tuple per cycle (doubled for the sacrificing). Each of these uses $n_{\mathcal{T}}$ initial bytes of randomness per input for the shares of a (and b). Furthermore, recall that each masked multiplication requires $\binom{n_{\mathcal{T}}}{2}$ bytes or randomness. That is, for $n_{\mathcal{T}} = 3$ and $m = 1$, we need 156 bytes of randomness per S-Box evaluation.

5.2.5 Cost Analysis and Scalability

The computation as described in §5.2.1 scales nicely with the masking order $n_{\mathcal{T}}$ and the security parameter m. For any fixed number of shares $n_{\mathcal{T}}$, the circuit area scales linearly in m (see for example Table 5.1). Storage increases with a factor $(m+1)n_{\mathcal{T}}$ compared to a plain implementation. We note that our implementations run in almost the same amount of cycles as that of a plain implementation. There is almost no loss in throughput and only negligible in latency. In software as well, the timing scales linearly if tiles run in parallel.

This efficiency does not come for free. The complexity is shifted to the preprocessing stage; indeed the generation of auxiliary triples is the most expensive part of the implementation. There is a trade-off to be made here between the online and offline complexity. The more auxiliary data we prepare "offline", the more efficient the online computation.

Complexity for Passive Attacker Scenario. It is remarkable that if active attackers are ruled out, and only SCA is a concern, then the complexity of the principal computation is linear in $n_{\mathcal{T}}$. This may seem like a significant improvement over previous masking schemes which have quadratic complexity on the security order [67, 108, 159]. However, this complexity is again pushed into the preprocessing stage. Nevertheless, this can be interesting especially for software implementations in platforms where a large amount of RAM is available to store the auxiliary data generated in §5.2.2. The same comments apply to FPGAs with plenty of BlockRAM.

Optimization of Preprocessing. It may be beneficial to store the output of the preprocessing stage §5.2.2 in a table for later usage. One could optimize this process by recycling auxiliary data (sample elements with replacement from the table). Of course, this would void the provable security claims; but if performed with care (with appropriate table shuffling and table elements refresh), this can give rise to an implementation that is secure in practice.

5.3 M&M

In this section, we present the second combined countermeasure, M&M, which evolves from CAPA by weakening the attacker model with the objective of finding a more optimal implementation. It is a new family of countermeasures that extends any SCA-secure masking scheme with information-theoretic MAC tags against DFA (*i.e.* **M**asks & **M**ACs) and combines them with an infective computation mechanism. We describe the M&M framework and its most fundamental building blocks, *i.e.* field multiplication and squaring in GF(2^k). We omit descriptions of trivial linear operations such as addition and scaling. Using these blocks, it is possible to secure any circuit against both SCA and DFA. Indeed, one only has to replace each AND-gate (resp. XOR-gate) with a M&M multiplication (resp. addition) in the field GF(2). We note that the proposed approach is only meant for the datapath. The control logic and public constants do not require a combined countermeasure as they are only vulnerable to FA.

Scheme Overview

We revisit the infective computation scheme of [131], which uses a redundant encryption of the plaintext and uses the difference between the two ciphertexts to infect the output. That is, if the ciphertexts match, the output is exactly

that ciphertext. If the ciphertexts do not match, the output is randomized so the attacker cannot get any information from it. The general idea is illustrated in Figure 5.5. For more details, we refer to the original work.

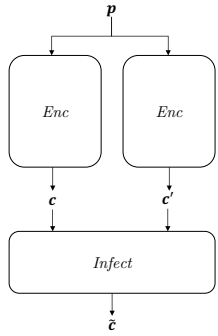


Figure 5.5: Infective Computation Scheme such as that of [131]

This scheme was broken in [22] because of a bias on the randomized output. We make two important changes. First, instead of using redundancy, which is vulnerable to the injection of identical faults, we replace the second instantiation of the cipher with a computation on information-theoretic MAC tags of the plaintext. If faults occur anywhere in the computation, the output of this block does not correspond to a valid MAC tag of the ciphertext with arbitrarily high probability. Second, we use the difference between what the MAC tag should be and what it actually is, to randomize the ciphertext without any bias. This is illustrated in Figure 5.6. We also ensure that one can find out whether the ciphertext is correct or not. In a way, we thus combine the advantages of detection and infection.

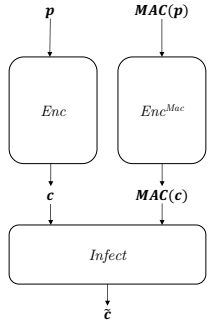


Figure 5.6: Our scheme

The computation on masks and MACs resembles to the approach of CAPA. However, instead of using expensive MPC machinery, we devise new constructions for generic field operations using existing SCA-secure gadgets and we port the information-theoretic MAC tags to the shared domain.

Adversarial Model

In this second countermeasure, we do not work within the full framework from the tile-probe-and-fault model. We consider a semi-invasive adversary with probing and faulting capabilities. On the one hand, we work under the d-probing model introduced in [108] for SCA, providing security against d^{th}-order side-channel analysis attacks under the independent leakage assumption. The model can include or exclude hardware glitches, but in this work, we specifically instantiate M&M considering glitches.

On the other hand, we consider two types of faults similarly to CAPA. The first faults are stochastic additive errors or ϵ-faults: this means the effect of a fault is the XOR of the current state with an error variable following some random distribution. The adversary is not limited in the number of bits he can alter, since we present a scheme which can tolerate *multiple* faults with *any Hamming weight*. In addition, we allow the attacker to inject non-stochastic faults (for example very precise laser injections or stuck-at faults). In that case however, the faults must be restricted to affect at most d of the $d + 1$ shares. We can justify this limitation by a proper placement of the circuit on the chip and the more complex setup of these kinds of faults.

In conclusion, on the one hand, the tile-probe part from the CAPA model is reduced to the d-probing model in M&M, which means the attacker can only probe d wires in the overall design instead of probing all the wires from all-but-one tile. On the other hand, the tile-fault part of the model is maintained, resulting in equivalent protection against faults.

The M&M Framework

In what follows, we describe how M&M extends an existing SCA-secure masking scheme with protection against faults. As already mentioned before, the literature provides us with many secure Boolean masking schemes to choose from [40, 98, 108, 147, 149, 153, 157]. Each of those is defined by how it performs non-linear operations (*i.e.* a multiplication) on Boolean shares. A specific instantiation of M&M thus depends on the choice of how to implement the shared multiplication operation $\vec{x} \odot \vec{y} = \vec{z}$. We assume that this operation

transforms $d+1$-sharings of two variables x and y into a $d+1$-sharing of their product $z = xy$. The latency and randomness cost of this operation depends on the choice of SCA countermeasure. However, we assume for now that the latency is one clock cycle, since this is the case in most schemes. For further discussion on the latency of the multiplication gadgets, see Section 5.3.4.

5.3.1 Basic M&M Building Blocks

M&M Multiplication. Given two operands $\langle \vec{x} \rangle$ and $\langle \vec{y} \rangle$, we want to compute the value shares $\vec{z}$ and tag shares $\vec{\tau}^z$ of $z = xy$. The value shares $\vec{z}$ can naturally be obtained using a shared multiplier with $\vec{x}$ and $\vec{y}$ as inputs. From this point, for ease of notation, we use $\overrightarrow{xy}$ to denote a sharing of the product xy.

$$\vec{x} \odot \vec{y} = \overrightarrow{xy} = \vec{z}$$

Deploying the same multiplier for the tag shares does not result in a valid tag for z

$$\vec{\tau}^x \odot \vec{\tau}^y = \overrightarrow{\alpha x} \odot \overrightarrow{\alpha y} = \overrightarrow{\alpha^2 z} \neq \vec{\tau}^z$$

However, if we use the result above in another shared multiplication with a sharing of α^{-1}, we can obtain the correct tag shares:

$$\overrightarrow{\alpha^{-1}} \odot (\vec{\tau}^x \odot \vec{\tau}^y) = \overrightarrow{\alpha z} = \vec{\tau}^z$$

Note that we could also obtain these tag shares by either $\vec{x} \odot \vec{\tau}^y$ or $\vec{\tau}^x \odot \vec{y}$, but we want to avoid crossing the datapaths of value and tag shares such that faults introduced in the values cannot automatically propagate to the tags. Consider for example a fault injected on input $\vec{x}$, resulting in $\vec{\tilde{x}}$. Then $\vec{\tilde{x}} \odot \vec{\tau}^y$ is a valid tag for $\vec{\tilde{x}} \odot \vec{y}$.

The M&M multiplication is summarized in the left side of Figure 5.7. We assume the operation $\odot$ includes one register stage, since this is the case for most state-of-the-art d-secure multipliers. The value-datapath of a M&M multiplication thus requires one clock cycle whereas that of the tags requires two.

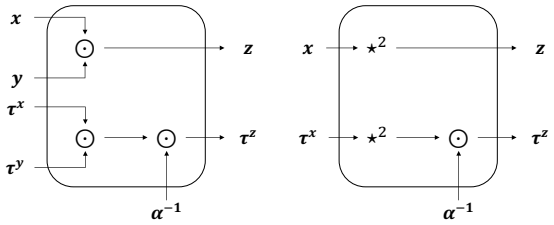


Figure 5.7: M&M non-linear operations: Obtaining the value and tag shares of $z = xy$(left) and $z = x^2$(right)

This multiplication uses a sharing of the inverse of the MAC key $\overrightarrow{\alpha^{-1}}$. We assume this is made available together with the sharing of the MAC key itself. If this is not the case, $\overrightarrow{\alpha^{-1}}$ can be precomputed and stored.

M&M Squaring. Note that squaring in M&M follows the same procedure. That is, to obtain $\langle \vec{z} \rangle$ from $\langle \vec{x} \rangle$ such that $z = x^2$, we first square the value shares $\vec{x}$ and tag shares $\vec{\tau}^x$. Since a characteristic-two finite field allows $(a+b)^2 = a^2+b^2$, squaring in the shared domain is a local operation that requires no registers: $\overrightarrow{x^2} = (x_0^2, \ldots, x_d^2)$ and $\overrightarrow{(\tau^x)^2} = ((\tau_0^x)^2, \ldots, (\tau_d^x)^2)$. We then again calculate the tag shares $\vec{\tau}^z$ by a shared multiplication with the inverse of the MAC key $\overrightarrow{\alpha^{-1}}$. The M&M squaring operation therefore takes one clock cycle and is depicted in the right side of Figure 5.7. The local squaring operation of shared data is depicted as $\star^2$. Extending this means that exponentiations by a power of two (x^{2^l}) take l clock cycles for the tags.

The Field GF(2)**.** Note that in the case of bits (the field GF(2)), no correction of the MAC tag is needed since we then have that $\alpha^2 = \alpha$, *i.e.* $\vec{\tau}^x \odot \vec{\tau}^y = \vec{\tau}^z$. As a result, the M&M multiplication in GF(2) has the same latency as the SCA-secure multiplication and M&M squaring (as well as exponentiation by a power of two) is local.

5.3.2 Building circuits with M&M

In this section we describe how to use the above building blocks to construct circuits for more complex operations in the M&M framework. Specifically, we

demonstrate the methodology for an inversion in $\mathrm{GF}(2^k), k > 2$,[2] which is of course of particular interest for implementing the AES S-Box. Furthermore, we introduce a method for processing an affine transformation over bits, as used as well in the AES S-Box.

Galois Field Inversion

We discuss two methods to do an inversion in Galois Field $\mathrm{GF}(2^k)$. The first constructs a multiplication chain from the M&M multiplication and squaring blocks described above. This methodology is generic and can be applied to any S-Box since any S-Box can be presented as a polynomial over the considered field. The second version uses a SCA-secure inversion implementation from existing literature to build a secure M&M block. This approach is specific for the AES S-Box, but results in a more efficient implementation.

Version 1: "Generic". For $x \in \mathrm{GF}(2^8)$, the inversion x^{-1} is equivalent to the power map x^{254}. We can obtain this function using the same power chain as before:

$$x^{254} = x^4 \cdot \left(\left(\left(x^5\right)^5\right)^5\right)^2$$

Since $x^5 = (x^2)^2 \cdot x$, this inversion requires seven M&M squares and four M&M multiplications. Using the above squaring and multiplication blocks, obtaining the inversion output thus requires 15 clock cycles.

Optimization 1. The calculation can be sped up using a specialized block for the exponentiation to the power five, shown in Figure 5.8. This is also done in [101] and justifies the choice of multiplication chain. In our case however, it is not trivial to do the same optimization for the tag calculation. The operation $\star^5$ raises the shares of x to the power five in one clock cycle. This requires a local computation of $\overrightarrow{x^4} = (x_0^4, \ldots, x_d^4)$, followed by a shared multiplication $\vec{x} \odot \overrightarrow{x^4} = \overrightarrow{x^5}$. This must be done with care since $\vec{x}$ and $\overrightarrow{x^4}$ are essentially the same variable and multiplying them may break non-completeness due to the dependencies among these two variables, as shown in previous chapter (cf. § 3.3.1). We therefore precompute and refresh $\overrightarrow{x^4}$ one cycle before it is used.

[2] In $\mathrm{GF}(2)$ and $\mathrm{GF}(2^2)$, inversion is trivial as it corresponds respectively to the identity and squaring function

After this first stage, which takes one clock cycle, we thus obtain a sharing of x^5 in the value datapath:

$$(x_0^4, \ldots, x_d^4) = \overrightarrow{x^4}$$

$$\vec{x} \odot \overrightarrow{x^4} = \overrightarrow{x^5}$$

and a sharing of $(\tau^x)^5 = \alpha^5 x^5$ in the tag datapath:

$$((\tau_0^x)^4, \ldots, (\tau_d^x)^4) = \overrightarrow{(\tau^x)^4}$$

$$\vec{\tau}^x \odot \overrightarrow{(\tau^x)^4} = \overrightarrow{(\tau^x)^5} \neq \vec{\tau}^{x^5}$$

A valid tag for x^5 is obtained through one more shared multiplication with $\overrightarrow{\alpha^{-4}}$, which is easily obtained locally from $\overrightarrow{\alpha^{-1}}$:

$$\overrightarrow{\alpha^{-4}} \odot (\vec{\tau}^x \odot \overrightarrow{(\tau^x)^4}) = \vec{\tau}^{x^5}$$

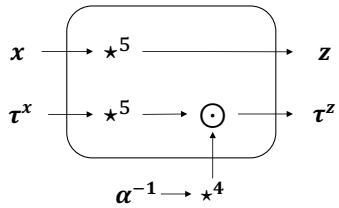


Figure 5.8: Obtaining the value and tag shares of $z = x^5$

With this specialized block, one obtains the exponentiation to the power five of $\langle \vec{x} \rangle$ in two clock cycles, if $(\overrightarrow{x^4}, \overrightarrow{(\tau^x)^4})$ is already refreshed beforehand. In those two clock cycles, we can obtain both the output $\langle \vec{z} \rangle$ and a refreshed $(\overrightarrow{z^4}, \overrightarrow{(\tau^z)^4})$ to be ready for the next block. The value shares can be refreshed using the second register stage (*i.e.* while the tag shares are being multiplied with α^{-4}). For the tag shares, there is no spare register stage for refreshing. In the shared multiplication with α^{-4}, we therefore raise each crossproduct to the power four. Before the register stage, we thus create a $(d+1)^2$-sharing of both τ^z and of $(\tau^z)^4$, each using its own randomness. As a result, the inversion result is available in only ten clock cycles:

- One cycle for the preparation of refreshed $\overrightarrow{x^4}$ and $\overrightarrow{(\tau^x)^4}$

- The calculation of $\langle\overrightarrow{x^{125}}\rangle = \langle\overrightarrow{\left(\left(x^5\right)^5\right)^5}\rangle$ requires six cycles.
- In the next cycle, we square $\langle\overrightarrow{x^{125}}\rangle$
- The last two cycles are spent on the multiplication of the result $\langle\overrightarrow{x^{250}}\rangle$ with $\langle\overrightarrow{x^4}\rangle$ to obtain $\langle\overrightarrow{x^{254}}\rangle$.

Optimization 2. By merging the last two operations into one step of two clock cycles, we reduce the total latency of the M&M inversion to nine cycles. This is possible because $x^{254} = f(x^4, x^{125})$ with $f(a,b) = a \cdot b^2$. We apply the same methodology as above. For the value shares:

$$(b_0^2, \ldots, b_d^2) = \overrightarrow{b^2}$$

$$\vec{a} \odot \overrightarrow{b^2} = \overrightarrow{f(a,b)}$$

For the tag shares:

$$((\tau_0^b)^2, \ldots, (\tau_d^b)^2) = \overrightarrow{(\tau^b)^2}$$

$$\vec{\tau}^a \odot \overrightarrow{(\tau^b)^2} = \overrightarrow{\alpha^3 f(a,b)} \neq \vec{\tau}^{f(a,b)}$$

$$\overrightarrow{\alpha^{-2}} \odot (\vec{\tau}^a \odot \overrightarrow{(\tau^b)^2}) = \vec{\tau}^{f(a,b)}$$

Figure 5.9 summarizes the nine-stage pipeline that calculates the value and tag shares for an inversion of $\langle\vec{x}\rangle$.

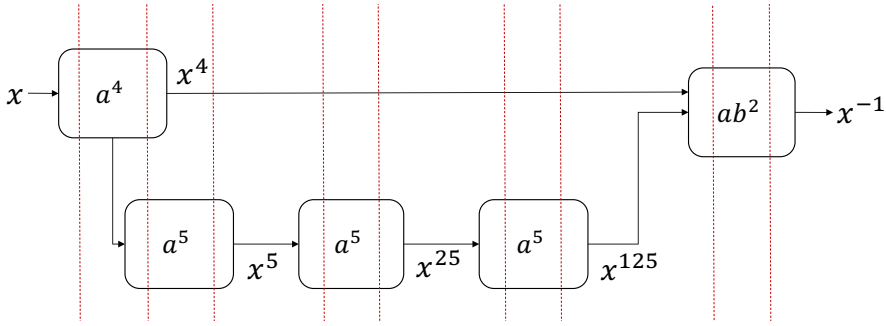


Figure 5.9: Inversion pipeline. (Register stages are depicted by red dotted vertical lines.)

Version 2: "Custom". The AES S-Box has been extensively studied in literature and an abundance of SCA-protected implementations has already been proposed. When implementing the AES S-Box in the M&M framework, it only makes sense to exploit the results from this research. We can take the above optimizations even further by merging all stages together into one. Consider applying a dth-order secure shared inversion in $\text{GF}(2^k)$ (denoted $\star^{-1}$) on both the value and tag shares. One obtains $\overrightarrow{x^{-1}}$ and $\overrightarrow{(\tau^x)^{-1}} = \overrightarrow{\alpha^{-1}x^{-1}} \neq \vec{\tau}^{x^{-1}}$. Only a shared multiplication with $\overrightarrow{\alpha^2}$ is required to calculate the correct tag shares of x^{-1}. This is illustrated in Figure 5.10. Again, it is easy to obtain $\overrightarrow{\alpha^2}$ by locally squaring the shares of α.

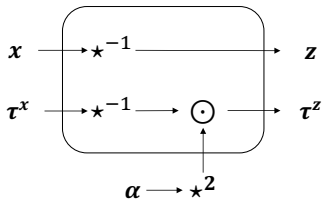


Figure 5.10: M&M inversion: Obtaining the value and tag shares of $z = x^{-1}$

Affine transformation over bits

The AES affine transformation at the end of the S-Box $A(x) = L(x) + c$ is linear over GF(2). The linear part of the transform, $L(x)$ is a matrix multiplication operating on the bitvector of x. A sharing of $L(x)$ is trivially obtained by applying the transform locally to each share of x. The same cannot be said for the tag shares of x. In this section, we describe how to obtain the tag shares for any linear transform of this type.

Isomorphisms. Consider the isomorphism ϕ between the finite field $\text{GF}(2^k)$ and the vector space $(\text{GF}(2))^k$, that maps each element to its bitrepresentation vector, *i.e.* $\phi(2^i) = \mathbf{e}_i$ with $\mathbf{e}_i$ the ith unit vector. We denote the bitvector of x as $\phi(x) = \mathbf{x}$. For the linear transform, we thus have

$$\phi(L(x)) = \mathbf{L}\phi(x) = \mathbf{L}\mathbf{x}$$

with $\mathbf{L} \in (\text{GF}(2))^{k\times k}$ the matrix that defines the linear transformation. One of the consequences of this isomorphism is

$$\forall \alpha \in \text{GF}(2^k), \exists \mathbf{M}_\alpha \in (\text{GF}(2))^{k\times k} \text{ s.t. } \forall x \in \text{GF}(2^k) : \phi(\alpha x) = \mathbf{M}_\alpha \phi(x) = \mathbf{M}_\alpha \mathbf{x}$$

Note that the opposite direction does not work: not for every $\mathbf{M} \in (\mathrm{GF}(2))^{k\times k}$ there exists an $\alpha \in \mathrm{GF}(2^k)$ such that this relation holds.

Given a value $x \in \mathrm{GF}(2^k)$ and a corresponding tag $\tau^x = \alpha x \in \mathrm{GF}(2^k)$, we wish to obtain a tag $\tau^{L(x)}$ satisfying $\tau^{L(x)} = \alpha L(x)$. We denote the bitvector of τ^x as $\phi(\tau^x) = \mathbf{t} = \mathbf{M}_\alpha \mathbf{x}$. We have

$$\begin{aligned}\phi(\tau^{L(x)}) = \phi(\alpha L(x)) &= \mathbf{M}_\alpha \phi(L(x)) \\ &= \mathbf{M}_\alpha \mathbf{L}\mathbf{x} \\ &= \mathbf{M}_\alpha \mathbf{L}(\mathbf{M}_{\alpha^{-1}}\mathbf{t}) \\ &= (\mathbf{M}_\alpha \mathbf{L}\mathbf{M}_{\alpha^{-1}})\mathbf{t}\end{aligned}$$

Hence, we can go from (x, τ^x) to $(L(x), \tau^{L(x)})$ by applying $\mathbf{L}$ to the bitvector of x and $\mathbf{M}_\alpha \mathbf{L}\mathbf{M}_{\alpha^{-1}}$ to the bitvector of τ^x. A similar approach is used in error detecting/correcting code schemes [48]. In our case however, the code depends on α and thus the matrix $\mathbf{M}_\alpha \mathbf{L}\mathbf{M}_{\alpha^{-1}}$ is secret and different in every encryption.

Calculating $\mathbf{M}_\alpha$. The matrix $\mathbf{M}_\alpha$ is straightforward to find. The ith column of $\mathbf{M}_\alpha$ can be denoted by $\mathbf{M}_\alpha \mathbf{e}_i = \mathbf{M}_\alpha \phi(2^i) = \phi(\alpha 2^i)$. We therefore know that

$$\mathbf{M}_\alpha = \begin{bmatrix}\phi(128\alpha) & \phi(64\alpha) & \dots & \phi(2\alpha) & \phi(\alpha)\end{bmatrix}$$

By seeing a matrix product as a compact way to describe k matrix-vector products

$$\mathbf{A}\mathbf{B} = \begin{bmatrix}\mathbf{A}\mathbf{b}_{k-1} & \mathbf{A}\mathbf{b}_{k-2} & \dots & \mathbf{A}\mathbf{b}_0\end{bmatrix}$$

we get that

$$\mathbf{L}\mathbf{M}_{\alpha^{-1}} = \begin{bmatrix}\phi(L(128\alpha^{-1})) & \phi(L(64\alpha^{-1})) & \dots & \phi(L(\alpha^{-1}))\end{bmatrix}$$

and thus

$$\mathbf{M}_\alpha \mathbf{L}\mathbf{M}_{\alpha^{-1}} = \begin{bmatrix}\phi(\alpha L(128\alpha^{-1})) & \phi(\alpha L(64\alpha^{-1})) & \dots & \phi(\alpha L(\alpha^{-1}))\end{bmatrix}$$

For each MAC key α, this matrix can be precomputed and stored in $d+1$ shares, similar to the precomputed sharing of α^{-1}. In the linear transformation, we obtain the tag shares $\vec{\tau}^{L(x)}$ by a shared matrix-vector multiplication with the sharing of $\mathbf{M}_\alpha \mathbf{L}\mathbf{M}_{\alpha^{-1}}$. A shared matrix-vector multiplication can use the same equations as the SCA-secure multiplier $\odot$, but with one of the inputs a matrix and with the field multiplication '$\cdot$' replaced by matrix-vector products. Because of the register stage in the shared matrix-vector multiplication, the affine transformation requires one clock cycle.

Latency Optimization. Note that in cases such as AES, where the affine transformation follows a power map, a little trick can ensure that the affine transformation does not increase the total latency of an S-Box evaluation. The last clock cycle of the inversion in Version 1 (resp. Version 2) is spent on a shared multiplication of intermediate tag shares with $\overrightarrow{\alpha^{-2}}$ (resp. $\overrightarrow{\alpha^{2}}$). We can incorporate this tag correction in the affine transformation by replacing the matrix $\mathbf{M}_{\alpha}\mathbf{L}\mathbf{M}_{\alpha^{-1}}$ with $\mathbf{M}_{\alpha}\mathbf{L}\mathbf{M}_{\alpha^{-3}}$ (resp. $\mathbf{M}_{\alpha}\mathbf{L}\mathbf{M}_{\alpha}$).

5.3.3 Infective Computation

We have described above how to implement two encryption blocks: one that calculates ciphertext shares, given plaintext shares and another that calculates ciphertext tag shares from plaintext tag shares and MAC key shares α_i. We now consider the ciphertext block-per-block with blocksize k. Let $c_i \in \mathrm{GF}(2^k)$ be the shares of one ciphertext block and $\tau_i^c \in \mathrm{GF}(2^k)$ the shares of the corresponding tag block. If the tags are consistent with the data, then $\sum_i \tau_i^c = \alpha \cdot \sum_i c_i$.

The Problem with Error Checking

The use of infective computation in M&M is motivated by the difficulty of designing an error checking mechanism that is secure against combined attacks. Consider the following algorithm that computes the error on the tags $E = \tau^c + \alpha \cdot c$ and verifies that it is zero. We assume sufficient registers are in place to prevent glitch problems.

Check$(\vec{c}, \overrightarrow{\tau^c}, \vec{\alpha})$
  Let $\vec{\theta} \leftarrow \vec{\alpha} \odot \vec{c}$
  for all shares i **do**
    Let $E_i \leftarrow \theta_i + \tau_i^c$
  $E = \sum_i E_i$
  Output $(E == 0)$

This checking algorithm is not secure in a combined adversary model with probing and faulting. When an attacker manages to insert a known fault Δ in one share of the shared multiplication such that the check is performed with

$\alpha' = \alpha + \Delta$, the unshared (zero) error E is replaced by

$$\begin{aligned} E' &= \tau^c + \alpha' \cdot c \\ &= \tau^c + \alpha \cdot c + \Delta \cdot c \\ &= \Delta \cdot c \end{aligned}$$

A single probe on the unshared E' thus reveals the (faulty) ciphertext c. This attack defeats the very purpose of the error check, which is to stop a faulty ciphertext from being released to the adversary.

The Solution

We propose the following routine Infect, which is local except for one shared multiplication to obtain a sharing of the correct MAC tag $\overrightarrow{\alpha c}$. Let R be a uniformly random mask $\in \mathrm{GF}(2^k) \setminus \{0\}$. Each share of the output ciphertext block is modified using this random mask and the difference of the tags.

Infect $(\vec{c}, \overrightarrow{\tau^c}, \vec{\alpha})$
  Let $\vec{\theta} \leftarrow \vec{\alpha} \odot \vec{c}$
  Draw $R \xleftarrow{\$} \mathrm{GF}(2^k) \setminus \{0\}$
  for all shares i **do**
    Let $\tilde{c}_i \leftarrow c_i + R \cdot (\theta_i + \tau_i^c)$
  Output $\vec{\tilde{c}}$

The scheme outputs a sharing of the adapted ciphertext block $\tilde{c} = \sum_i \tilde{c}_i = c + R \cdot (\alpha \cdot c + \tau^c)$. Thus, if the tags are consistent ($\alpha \cdot c + \tau^c = 0$), the scheme outputs a sharing of the computed block c. On the other hand, if the tags do not match ($\alpha \cdot c + \tau^c \neq 0$), the unshared output $\tilde{c}$ is random.

One may note that generating a nonzero mask R is nontrivial. However, there must be a PRNG with enough throughput to realize all the randomness for the computation of the S-Box. The number of random bits available for the routine Infect is thus much higher than the amount required. From this, it is easy to generate one nonzero byte.

Unbiased Randomization We verify that the infected ciphertexts are uniformly distributed, so the attacker cannot obtain any information from them. Consider the case when the computation of Enc (and Enc^{Mac}) is disturbed by

faults Δ^c (resp. Δ^τ), resulting in the unshared infected ciphertext block

$$\tilde{c} = c + \Delta^c + R \cdot (\alpha \cdot (c + \Delta^c) + \tau^c + \Delta^\tau)$$

We may assume Δ^c is non-zero and unkown to the attacker. In fact, knowing the faulty ciphertext, or indeed Δ^c, is the goal of the adversary in a DFA attack. Furthermore, we assume a strong probing adversary can know the value of the mask R, which is why we do not allow it to be zero. However, the MAC key α is always secret due to sharing. These introduced faults (Δ^c, Δ^τ) remain undetected with a probability of 2^{-km}, corresponding to the case when $\Delta^\tau = \alpha \cdot \Delta^c$. We therefore claim an error detection probability (EDP) of $1 - 2^{-km}$ and focus now on the case when $\Delta^\tau \neq \alpha \cdot \Delta^c$.

Using the fact that $\alpha \cdot c + \tau^c = 0$, we rewrite $\tilde{c} = c + \Delta^c \cdot (1 + R \cdot \alpha) + \Delta^\tau \cdot R$. Clearly, the unbiased randomization of the output depends on the uniformity of the mask $(1 + R \cdot \alpha)$. It can be verified that this mask is uniformly random in $\mathrm{GF}(2^k)$ when R is uniformly random in $\mathrm{GF}(2^k) \setminus \{0\}$ and α uniformly random in $\mathrm{GF}(2^k)$. As a result, $\tilde{c}$ is uniformly random in $\mathrm{GF}(2^k)$ when $\Delta^c \neq 0$.

Combining Infection and Detection

In many applications, outputting garbage when the chip is under attack suffices. There are however some use cases, where one might want to know whether the outputted ciphertext is correct. If that is the case, we propose to do the infective computation twice, with different masks and with different MAC keys. The two resulting ciphertexts can be compared: if the computation was corrupted, the ciphertexts are distinct and randomized; otherwise, they are identical.

The above adaption requires that the number of MAC keys m is at least two. If $m = 1$ suffices for security, we propose the following solution to avoid duplicating the tagsize just for the sake of outputting two ciphertexts. A second MAC key β is created only for the infective computation part and not for the cipher evaluation. The procedure Infect_2 below describes how to obtain the second ciphertext block $\vec{\tilde{c}'}$ apart from the original $\vec{\tilde{c}}$ obtained with Infect.

$\mathrm{Infect}_2\ (\vec{c}, \vec{\tau^c}, \vec{\alpha^{-1}}, \vec{\beta}, R)$
  Let $\vec{\theta'} \leftarrow \vec{\beta} \odot \vec{c}$
  Let $\vec{\tau^{c'}} \leftarrow \vec{\alpha^{-1}} \odot (\vec{\beta} \odot \vec{\tau^c})$
  Draw $R' \xleftarrow{\$} \mathrm{GF}(2^k) \setminus \{0, R\}$
  for all shares i **do**
    Let $\tilde{c}'_i \leftarrow c_i + R' \cdot ({\theta_i}' + \tau_i^{c'})$
  Output $\vec{\tilde{c}'}$

The two unshared outputs are thus

$$\tilde{c} = \sum_i \tilde{c}_i = c + R \cdot (\alpha \cdot c + \tau^c)$$

$$\tilde{c}' = \sum_i \tilde{c}'_i = c + R' \cdot (\beta \cdot c + \beta \cdot \alpha^{-1} \cdot \tau^c)$$

5.3.4 AES Case Study

M&M has been designed in a way that allows it to use both provably secure (*e.g.* SNI [18]) gadgets or more efficient but non-provably secure blocks. We now present specific AES implementations confirming the latter. We target first- and second-order security as those are relevant for realistic attacks and use TVLA for these specific orders to demonstrate the security.

Our implementations are mere examples of the many different ways to instantiate M&M. Any existing or future SCA-secure gadgets can be used as the underlying building blocks. In this section, we first detail our choice of gadgets and investigate the implementation cost of the resulting AES constructions. We then empirically validate our SCA claims using univariate and bivariate test vector leakage assessment and we perform a simulation-based verification of the DFA security.

Implementation Details

In § 5.3.2, we described essentially all components that are needed to construct an AES SubBytes implementation: the inversion in GF(2^8) and the affine transformation over bits. We distinguish two versions of the S-Box implementation. One follows a rather generic methodology using multiplication chains and the other is customized for the inversion. We investigate the implementation cost of both implementations for first- and second-order SCA security ($d = 1$ or 2). For our specific instantiations of M&M, we only claim first- and second-order security. Nevertheless, it is extendable to higher-orders given a suitable choice of building blocks.

The remaining AES blocks (ShiftRows, MixColumns and AddRoundKey) consist exclusively of linear operations and are thus trivially implemented for the M&M framework. More specifically, the datapaths in both the Enc and Enc^{Mac} blocks each consist of $d + 1$ copies of the AES state and key arrays. These arrays contain data shares in the encryption block Enc and tag shares in case of the Enc^{Mac} block. In total, the area cost of these blocks increases with a factor

$(d+1)(m+1)$ compared to an unprotected implementation. Our implementation uses the same byte-serialized architecture as used in [98].

We now detail our choice of multiplication and refresh gadgets which are used in the multiplication chain version of the inversion (cf. version 1, Figure 5.9) and the inversion gadget used in version 2 (cf. Figure 5.10).

Multiplication gadgets. We choose a $d+1$-share multiplier as our SCA-secure multiplier. The following equations describe for example a three-share, second-order ($d=2$) secure multiplication of shared variables $\vec{x} = (x_0, x_1, x_2)$ and $\vec{y} = (y_0, y_1, y_2)$ into $\vec{z} = (z_0, z_1, z_2)$, using three units of randomness $r_0, r_1, r_2 \in \mathrm{GF}(2^k)$ as in [98]. We first calculate nine intermediate values t_{ij} for $i, j \in \{0, 1, 2\}$. After a register stage for synchronization of t_{ij}, we compute the output shares $\vec{z} = (z_0, z_1, z_2)$. The latency of the operation $\odot$ is thus 1 clock cycle.

$$t_{00} = x_0 \cdot y_0$$

$$t_{01} = x_0 \cdot y_1 + r_0$$

$$t_{02} = x_0 \cdot y_2 + r_1$$

$$t_{10} = x_1 \cdot y_0 + r_0 \qquad z_0 = [t_{00}]_{reg} + [t_{01}]_{reg} + [t_{02}]_{reg}$$

$$t_{11} = x_1 \cdot y_1 \qquad z_1 = [t_{10}]_{reg} + [t_{11}]_{reg} + [t_{12}]_{reg}$$

$$t_{12} = x_1 \cdot y_2 + r_2 \qquad z_2 = [t_{20}]_{reg} + [t_{21}]_{reg} + [t_{22}]_{reg}$$

$$t_{20} = x_2 \cdot y_0 + r_1$$

$$t_{21} = x_2 \cdot y_1 + r_2$$

$$t_{22} = x_2 \cdot y_2$$

The corresponding first-order construction is given in [98]. Each multiplication requires $\binom{d+1}{2}$ fresh units of randomness. It has been shown in [89] that such a multiplication gadget is composable if the result is stored in a register. As shown on Figure 5.9, we present in this work a construction with such registers in the value share datapath but *without* extra registers in the tag share datapath. Note that, the intermediate registers of [89] are a requirement for SNI schemes in HW, but the SNI property is not a prerequisite for a scheme to be secure and

we customized our design for better performance. We see currently no formal way to prove the security of our construction, but because the tag shares can be seen as Boolean shares of a multiplicative share of the secret, we conclude that the tag shares do not need to be stored in registers in order to obtain security. We verify our approach empirically using TVLA and do not detect any leakage (cf. §5.3.4). Provable security can be achieved by adding registers to each of the tag share and value share datapaths.

Refreshing gadgets. When $d = 1$, a $d + 1$-share variable $\vec{x}$ can be refreshed with d fresh random units using additive refreshing:

$$(x_0, x_1, \ldots, x_d) \to (x_0 + r_0, x_1 + r_1, \ldots, x_{d-1} + r_{d-1}, x_d + \sum_{i=0}^{d-1} r_i)$$

For second-order security, we use ring refreshing as in [65], which consumes $d + 1$ fresh random units.

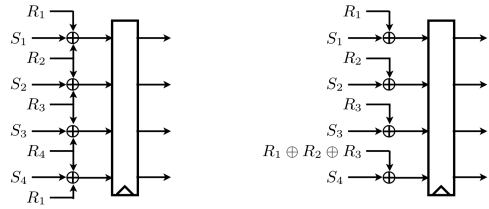


Figure 5.11: Ring and Additive Refreshing [65]

We use these refreshings for the shares of x^4, x^{20} and x^{100} in Figure 5.9 as well as for the shares of $(\tau^x)^4$. We let $r(d)$ be the corresponding randomness cost for refreshing $d + 1$ shares with security order d, *i.e.* $r(1) = 1$ and $r(2) = 3$. The shares of $(\tau^{x^5})^4$ and $(\tau^{x^{25}})^4$ are refreshed using $\binom{d+1}{2}$ units during the last multiplication in the M&M power five block (cf. Figure 5.8). For $d \in \{1, 2\}$, we actually have that $r(d) = \binom{d+1}{2}$, so the cost of each refreshing is $r(d)$. We note again that these types of refreshing gadgets are only to be used for respectively first- and second-order security and are not secure for higher orders.

Inversion gadget. For the shared inversion in GF(2^8), $\star^{-1}$, we can use De Cnudde *et al.*'s $d + 1$ [65] or Gross *et al.*'s Domain Oriented Masking [98] AES S-Box implementations. Both are based on Canright's compact S-Box [57] using the tower field approach. We opt for the first, which requires five register stages.

Table 5.6: Randomness Cost for the AES S-Box implementations

	# $\odot$	# $\star^{-1}$	# Fresh	# Random Bits		
				d	$d=1$	$d=2$
Shared Mult. ($\odot$)/Refresh	1	-	-	$8\binom{d+1}{2}$	8	24
Shared Inv. ($\star^{-1}$) [65]	-	1	-	-	54	162
Fresh Mask	-	-	1	8	8	8
S-Box V1	14+6	-	-	$160\binom{d+1}{2}$	160	480
S-Box V2	1	2	-	-	116	348
Infective Computation	1	-	1	$8\binom{d+1}{2}+8$	16	32

Together with the final shared multiplication with α^2, the latency of the M&M inversion in Figure 5.10 is thus six cycles.

Implementation Cost

Randomness. We summarize the randomness cost in Table 5.6.

Recall that for $d \in \{1, 2\}$, each shared multiplication $\odot$ and each refreshing consume $\binom{d+1}{2}$ units of randomness, where each unit is a byte in the case of AES. The inversion pipeline of Figure 5.9 performs four M&M non-linear operations (three times a^5 and once ab^2). Each of these requires exactly three $\odot$'s, although we count one less for the $f(a, b) = ab^2$ block (because this operation is included in the affine transform). We count two additional $\odot$'s with $\overrightarrow{\alpha^{-1}}$ for the computation of the tag shares of $\overrightarrow{x^4}$ and one $\odot$ for the tag shares in the affine transform. This brings the total number of shared multiplications $\odot$ to 14. In addition, we need 6 refreshings to preserve the non-completeness in the power five exponentiations $\star^5$.

The inversion circuit from [65] consumes 54 (resp. 162) bits of randomness in first-(resp. second-)order. The M&M inversion in Figure 5.10 uses this circuit twice. Furthermore, the affine transformation adds one additional shared (matrix) multiplication.

Finally, the infective computation uses randomness for one shared multiplication and an additional unit for the mask R. However, since the infection takes place when all SubBytes evaluations have finished, the total randomness does not increase.

Latency. The first inversion from §5.3.2 requires nine clock cycles. In the second version, we use the $\star^{-1}$ implementation from [65], which results in an

M&M inversion of six clock cycles. Recall from §5.3.2 that the AES affine transformation in M&M requires no additional cycles. In total, the AES S-Box output is thus obtained in nine clock cycles with version 1 and six clock cycles with version 2.

The byte-serialized architecture from [98] is very efficient as it performs the MixColumns, ShiftRows and AddRoundKey stages in parallel with the SubBytes stage. As a result, when the S-Box latency is $C \geq 4$ cycles, one round of encryption (including key schedule) requires exactly $16 + C$ clock cycles: During the first 16 cycles, all the state bytes are fed to the S-Box pipeline input. The MixColumns operation is done in parallel every four cycles. The remaining C cycles are spent waiting for the last S-Box output. During the first four of these, the S-Box can be used by the key schedule. In the last cycle, the last S-Box output is shifted into the state at the same time as ShiftRows is performed.

Our two versions of the AES Encryption therefore require respectively 25 and 22 clock cycles per encryption round.

Area. We report our area results for first- and second-order security in Table 5.7 together with latency and randomness cost. As expected, the more customized version of the S-Box results in a much more efficient implementation. Note that many more tradeoffs between area, latency and randomness cost are possible depending on design choices (*e.g.* multiplication chain) and used building blocks (*e.g.* shared inversion block $\star^{-1}$).

Table 5.7: Cost of first- and second-order AES implementations

	Area [kGE]		**Latency**	**# Random bits/cycle**	
	$d = 1$	$d = 2$	**[# cycles]**	$d = 1$	$d = 2$
State Array	5.6	8.5	-	-	-
Key Array	4.2	6.3	-	-	-
S-Box					
• V1	19.2	42.0	9	160	480
• V2	6.5	13.5	6	116	348
Control	0.2	0.2	-	-	-
Infective Comp.	1.7	3.3	-	16	32
Other	1.0	1.4	-	-	-
Total					
• V1	31.9	61.7	266	160	480
• V2	19.2	33.2	236	116	348

Comparison to State-of-the-art. In Table 5.8, we report our area results next to other state-of-the-art schemes. Some of these protect only against SCA [65, 98] and some are combined countermeasures, such as ParTI [168] and CAPA (cf. § 5.2). For the latter, it is not easy to compare the results given the difference in cipher implemented and synthesis libraries used. We try to overcome these differences by also reporting the overhead factor of the combined countermeasure, compared to an implementation that provides only protection against SCA.

The ParTI countermeasure is applied to the LED scheme in [168]. The authors report an area of 20.2 kGE obtained with a UMC 0.18μm library [105], compared to 7.9 kGE for the SCA-only first-order secure LED implementation. This signifies an area overhead factor of $\frac{20.2}{7.9} = 2.56$. For the combined countermeasure CAPA, we can compute the overhead over a SCA-secure KATAN implementation from Tab. 5.1. Finally, we compare our first-order M&M AES (V2) with De Cnudde's [65] first-order implementation against SCA only. Table 5.8 reports the area of the implementation that is only secure against SCA in the fourth column and the area of the combined countermeasure in the fifth column. The overhead factor for those can be found in the last column. We note that the dependency on the synthesis library cannot completely be eliminated this way. Furthermore, all schemes consider very different adversary models.

Table 5.9 does the same for second-order secure implementations.

Table 5.8: Area comparison for first-order secure implementations

Countermeasure	Synthesis Library	Cipher	SCA-only [kGE]	Combined [kGE]	**Overhead factor**
[65]	Nangate 45nm [144]	AES	7.6[3]	-	-
[98]	UMC 90nm Low-K	AES	6.0	-	-
CAPA	Nangate 45nm [144]	KATAN	3.6	30.5	**8.47**
ParTI [168]	UMC 0.18μm [105]	LED	7.9	20.2	**2.56**
M&M	NanGate 45nm [144]	AES	7.6	19.2	**2.53**

Evaluation

Evaluation of our M&M implementations is done separately for SCA and DFA, since no comprehensive method for verifying against combined attacks has been published to our knowledge. For the first, we program both versions of a second-order protected AES on an FPGA and evaluate the leakage coming

[3]This number differs from the one reported in [65]. We synthesized the original implementation with the same software and library as our M&M implementation.

Table 5.9: Area comparison for second-order secure implementations

Countermeasure	Synthesis Library	Cipher	SCA-only [kGE]	Combined [kGE]	**Overhead factor**
[65]	Nangate 45*nm* [144]	AES	12.6[1]	-	-
[98]	UMC 90nm Low-K	AES	10.0	-	-
CAPA [158]	Nangate 45*nm* [144]	KATAN	5.9	55.2	**9.35**
M&M	NanGate 45*nm* [144]	AES	12.6	33.2	**2.63**

from the power consumption with a non-specific t-test. The state-of-the-art on evaluating fault countermeasures is less advanced. We evaluate the scheme's EDP through a simulation of the circuits, in which we model additive faults in the RTL.

SCA evaluation

We first perform the t-test on an unprotected AES implementation to verify that our setup is sound and able to detect leakage. We emulate the unprotected implementation by disabling the PRNG. Leakage is then expected in every order. When we turn the PRNG on and activate the countermeasures, we expect only third-order leakage since the implementation is second-order secure.

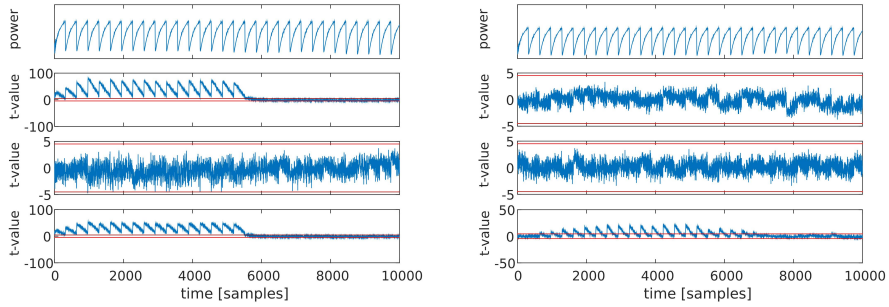


Figure 5.12: Non-specific t-test on second-order secure M&M AES implementation, V1. Left: PRNG off (24K traces); Right: PRNG on (100M traces). Rows (top to bottom): one exemplary power trace; first-order t-test; second-order t-test; third-order t-test.

The t-test results for version 1 and 2 of our AES implementation are shown in Figures 5.12 and 5.13 respectively. In both cases, we see clear evidence of leakage at only 24 000 traces when the PRNG is turned off. When we enable the

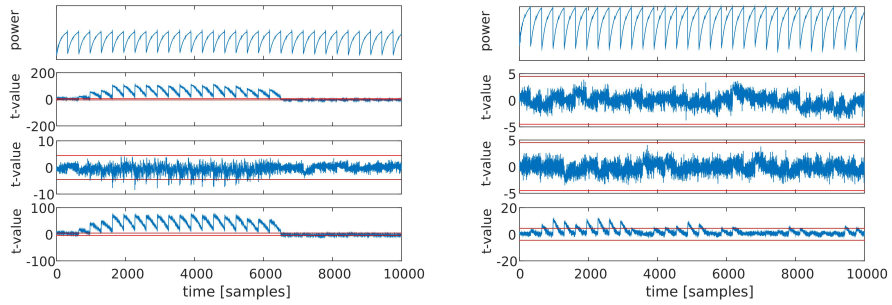


Figure 5.13: Non-specific t-test on second-order secure M&M AES implementation, V2. Left: PRNG off (24K traces); Right: PRNG on (100M traces). Rows (top to bottom): one exemplary power trace; first-order t-test; second-order t-test; third-order t-test.

PRNG, neither the first- nor second-order t-test statistics surpass the threshold 4.5 with up to 100 million power traces.

In addition, we perform a bivariate analysis by combining time samples. For memory efficiency, we reduce the resolution of the oscilloscope to 100MS/s, resulting in power traces of 1 000 time samples each. We then perform the t-test as described above on $1\,000 \times 1\,000$ matrices, formed by a centered product of the traces. The results are shown in Figure 5.14 and confirm that there is no bivariate leakage with up to 50 million traces.

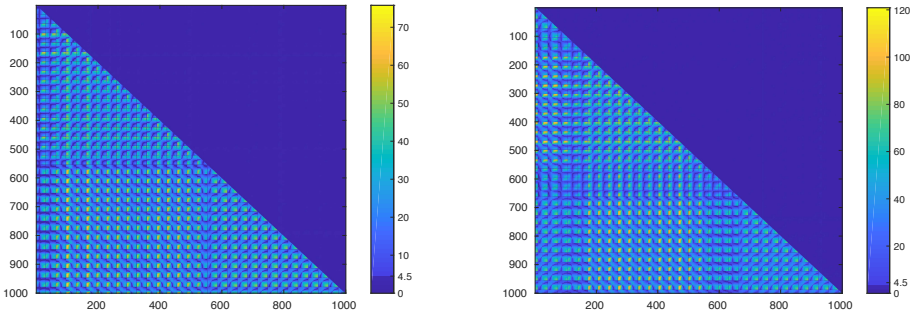


Figure 5.14: Bivariate t-test on second-order secure M&M AES imlementation, V1 (left) and V2 (right). Below diagonal: PRNG off (20K traces); Above diagonal: PRNG on (50M traces).

FA evaluation

In this section we evaluate the behaviour of our design when *multiple* stochastic faults are injected. We describe our experiment, which aims to measure M&M's detection rate of faults. For simplicity, we evaluate our first-order secure AES implementations.

Fault modeling. Traditionally, fault modeling theory distinguishes between faults that affect the logic function on the one hand and delay faults on the other. Moreover, faults can be clasified as structural (modifying the interconnections among components in the circuit) or functional (modifying the functionality of certain parts of the circuit) [3].

Faults in cryptographic devices are typically injected with a laser or introduced by clock or power line glitches. In our experiments we consider functional faults in the logic functions to model the adversary of §5.3 (*i.e.* additive errors). We model faults using XOR additions. This does not only allow us to flip one specific bit, but also to XOR entire offsets to k-bit words.

Fault injection. We enable a fault injection on a wire by extending the original VHDL code with an additional *fault gate* on that wire. Such a gate is simply an XOR with a *fault selector*, indicating whether or not we want to inject a fault.

In each design to be tested, we select a number of critical bytes where an attacker is most likely to inject a fault. These points are: the input to the state register; the state and key byte before AddRoundKey; the SubBytes input from the key schedule and four different points inside SubBytes. Faults can be inserted in every data share and every tag share of those bytes.

This means that 256 fault gates are installed in the first order implementations. We collect the corresponding fault selectors in a fault vector, which is controlled by the testbench. Each bit set to '1' in the fault vector corresponds to a fault on a single bit. By setting multiple bits in the vector, we enable *multiple faults* in the implementation. When several faults are activated in the same byte, it implies the XOR of an offset to that variable.

We want to be able to randomly draw fault vectors with a chosen Hamming weight H. For this, we draw inspiration from the basic principles of address decoding. We draw one random bit; if it is 'zero', a fault occurs in the first half of the vector and if it is 'one', in the second half. We draw a second bit and follow the same procedure to decide which of the two quarters. We continue this way until a single fault bit is selected. Thus, $log_2(256) = 8$ random bits

are needed to set a one-bit fault in a 256-bit vector. By repeating this method H times, we can draw random fault vectors with Hamming weight H. In our experiments we choose $H = 128$.

For each selected bit, we flip one more coin that decides whether the selected fault is activated or not. This means that, of the 128 selected faults, approximately half will be active. Since most of the fault attacks in literature target one of the last rounds of AES, we similarly "inject" our faults in the last round of encryption.

Results. We simulate the fault-augmented VHDL code with Xilinx ISIM for 50 000 iterations and measure the fault detection rate. In each experiment, approximately 64 bits are altered in the computation. We are thus faulting one or more bits in multiple bytes. We consider the faults *detected* if the returned ciphertext is infected. In version 1 of our M&M AES, the experiment shows that 210 faulty ciphertexts are not infected. This means that the resulting rate of detection from the experiments of our M&M implementation is 0.9958, compared to the theoretical $1 - 2^{-8} = 0.9961$. In our second AES version, 189 faulty ciphertexts are not infected, which means the detection probability from the experiments is 0.9962.

5.4 Security Claims and Discussion

In this section, we discuss the security of CAPA and M&M schemes against SCA, FA, and combined attacks. Then, we discuss how both schemes perform against known attacks from the literature. Subsequently, we provide further information to take into account regarding the MAC tags, and finally, we comment on the differences between CAPA and the SPDZ protocol.

5.4.1 Security Analysis CAPA

With both described adversaries $\mathscr{A}_1$ and $\mathscr{A}_2$, our design CAPA claims provable security against the following types of attacks, as well as a combination of them:

1. Side-Channel Analysis (*i.e.* against $n_{\mathcal{T}} - 1$ tile probing adversary).
2. Fault Attacks (*i.e.* an adversary introducing either known faults into $n_{\mathcal{T}} - 1$ tiles or random faults everywhere).

Security against SCA

One can check that no union of $n_{\mathcal{T}} - 1$ tiles $\cup_{j \in j_1,\ldots,j_{n_{\mathcal{T}}-1}} \mathcal{T}_j$ has all the shares of a sensitive value. Very briefly, we can reason to this $(n_{\mathcal{T}} - 1)^{\text{th}}$-order non-completeness as follows. All computations are local with the exception of the unmasking of public values such as ε. However, the broadcasting of all shares of ε does not break non-completeness since $\varepsilon = x + a$ is not sensitive itself but rather a blinded version of a sensitive value x, using a random a that is shared across all tiles. Unmasking the public value ε therefore gives each tile $\mathcal{T}_i$ only one share $\varepsilon + a_i$ of a new sharing of the secret x:

$$\vec{x} = (a_1, \ldots, a_{i-1}, \varepsilon + a_i, a_{i+1}, \ldots, a_{n_{\mathcal{T}}})$$

In this sharing, no union of $n_{\mathcal{T}} - 1$ shares suffices to recover the secret. Our architecture thus provides non-completeness for all sensitive values. As a result, our $n_{\mathcal{T}}$-share implementation is secure against $n_{\mathcal{T}} - 1$-probing attacks. Any number of probes following the adversaries' restrictions leak no sensitive data. Our model is related to the wire-probe model, but with wires replaced by entire tiles. We can thus at least claim security against $(n_{\mathcal{T}} - 1)^{\text{th}}$-order SCA.

Security against FA

A fault is only undetected if both value and MAC tag shares are modified such that they are consistent. Adversary $\mathcal{A}_1$ can fault at most $n_f < n_{\mathcal{T}}$ tiles, which means he requires knowledge of the MAC key $\alpha \in GF(2^{km})$ to forge a valid tag for a faulty value. Since α is secret, his best option is to guess the MAC key. This guess is correct with probability 2^{-km}. Adversary $\mathcal{A}_2$ has ϵ-faulting abilities only and will therefore only avoid detection if the induced faults in value and tag shares *happen* to be consistent. This is the case with probability 2^{-km}. We can therefore claim an error detection probability (EDP) of $1 - \frac{1}{2^{km}}$. The EDP does not depend on the number of faulty bits (or the hamming weight of the injected fault).

Security against Combined Attacks

In a combined attack, an adversary with n_f-faulting capabilities can mount an attack where he uses the knowledge obtained from probing some tiles $\in \mathcal{P}_1$ to carefully forge the faults. In SPDZ, commitments are used to avoid the so called "rushing adversary", which first collects all inputs from the other parties and only then decides what to send in reply. CAPA does not need commitments as the timing limitation on $\mathscr{A}_1$ adversary ensures a n_f-fault cannot

be preceded by a probe in the same clock cycle. As a result, we inherit the security claims of SPDZ and the claimed EDP is not affected by probing or SCA. Also, the injection of a fault in CAPA does not change the side-channel security. Performing a side-channel attack on a perturbed execution does not reveal any additional information because the Beaver operations do not allow injected faults to propagate through a calculation into a difference that depends on sensitive information. We can claim this security, because of the aspects inherited from MPC. CAPA is essentially secure against a very powerful adversary that has complete control (hence *combined* attacks) over all but one of the tiles.

5.4.2 Security Analysis M&M

Security against SCA

M&M can be based on several different Boolean masking schemes providing SCA secure multiplication, inversion and refreshing gadgets in the chosen adversary model, and thus the security of any instantiation depends heavily on those choices.

By adhering to security principles such as ensuring non-completeness [40] and proper refreshing is satisfied everywhere, M&M inherits the SCA security of the shared multiplication and inversion mechanism used. A Boolean masking scheme that is secure in the considered SCA attacker model thus provides security against d^{th}-order SCA. Since the model can include or exclude hardware glitches if the shared multiplication and inversion mechanism used are also secure in the presence of glitches, M&M inherits this.

The computation of the tag shares follows the same design principles as the value share calculations. The two datapaths operate completely independently of each other and receive their own distinct fresh randomness (see Figure 5.15). It is important to note that the input sharings $\vec{p}$ and $\overrightarrow{\tau^p}$ must be independent as well, which is easily achieved if the initial maskings of p and τ^p are obtained separately. The independence of the two datapaths ensure that their merging in the *Infect* block does not induce leakage on p or τ^p.

The Refreshing Gadget. It is important that any refreshing mechanism used (cf. § 5.3.2) ensures the same security as is provided by the used masking scheme. The kind of refreshing thus depends on the targeted security order d [27] and the considered attacker model. In general, one can always make use of the multiplication-based refresh gadget of Ishai *et al.* [108]. It has been shown in [19, Gadget 4b] that this refreshing ensures composability at any order. For

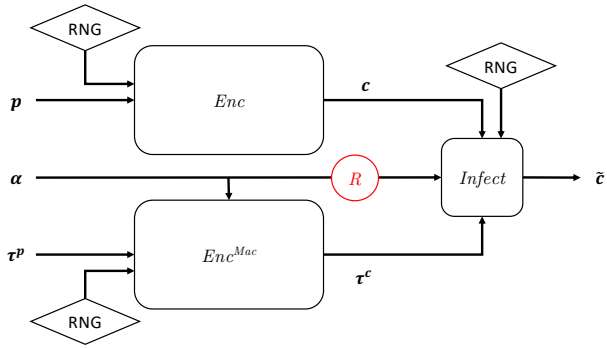


Figure 5.15: Overview of the scheme

a specific target security level, randomness can be consumed more efficiently. For example, the *ring refreshing* approach of [65] uses $d+1$ fresh masks in a circular manner to refresh $d+1$ shares. This method suffices for second-order security. At certain higher orders, one can use its variant, *offset refreshing*, which still uses only $d+1$ units of fresh randomness but rotates with an offset of more than 1 [20, Alg. 2]. Finally, *additive refreshing* using only d fresh masks is sufficient when first-order security is targeted. For a more detailed treatment of refreshing gadgets, we refer to [20].

Security against FA

The introduction of MAC tags to the circuit results in resistance against FA, in which faults are not limited in Hamming weight nor in quantity. Faults are only undetected if both the value and MAC tag shares are modified in such a way that the relation

$$\sum_i \tau_i^x = \alpha \cdot \sum_i x_i \tag{5.2}$$

remains true. Since the MAC key $\alpha \in \mathrm{GF}(2^{km})$ is unkown to the adversary, any number of stochastic additive errors result in this relation with probability at most 2^{-km}. If the attacker has the ability to inject non-stochastic faults, our model restricts these to affect at most d of the $d+1$ shares. In that case, the success probability still depends on the probability of guessing the secret MAC key $\alpha \in \mathrm{GF}(2^{km})$ correctly. We therefore claim an error detection probability (EDP) of $1-2^{-km}$. Note that, if α were fixed and identical for all encryptions, the values and tags would be equivalent to an error detecting code.

As stated in the adversarial model in §5.3, the faults we consider are neither limited in Hamming weight nor in quantity. The worst-case probability that satisfies(5.2) is 2^{-km} regardless of the Hamming weight of a single fault. The accumulated effect of multiple faults also does not change this probability. The adversary obtains no additional information after injecting faults, hence random shooting or guessing α remains the best strategy for subsequent faults. In the end, the same equation (5.2) in $\mathrm{GF}(2^{km})$ must hold for the faults to remain undetected. It holds with probability 2^{-km}. We experimentally investigate the effect of multiple faults in Section 5.3.4.

Security against Combined Attacks

Having brought the MAC tags to the shared domain, our scheme also provides security against combined attacks. Thanks to the fact that the MAC key α is shared, it remains secret even to a probing adversary. As a result, the detection probability of injected faults does not change, even when the adversary combines them with SCA. We recall that our model limits the injection of deterministic faults to d of the $d+1$ shares. In case of a combined attack, the total number of affected shares by either faults or probes should thus still not exceed d.

Modern combined attacks such as PACA [6,61], which require a faulty ciphertext to succeed, are prevented since faulty ciphertexts are only released with probability 2^{-km}. The effective complexity of such attacks thus increases by a factor at least 2^{km}. Thanks to the infective computation, M&M is also secure against combined attacks that target the checking mechanism or that exploit correlations on the faulty ciphertext [73, 161].

Although we are not aware of any combined attack against M&M, we cannot formally prove it as with CAPA. The only provable approach against combined attacks known so far is to adapt an actively secure MPC protocol. No other formal techniques have yet been found.

5.4.3 Attacks

The Glitch Power Supply or Clock Attack. The solutions presented in this chapter critically depend on the fact that there is no single point where an attacker can insert a fault that affects all $n_{\mathcal{T}}$ tiles (CAPA) or all $d+1$ shares (M&M) deterministically. An attacker may try to glitch the chip clock line that is shared among all tiles (resp. shares). In this case, the attacker could try to carefully insert a glitch so that writing to the abort register is skipped or bypass the infection routine. Since all tiles (resp. shares) share the same clock,

the attacker can bypass in this way the tag verification step (resp. the infection step). Similar comments apply, for example, to glitches in the power line. The bottom line is that one should design the hardware architecture accordingly, that is, deploy low-level circuit countermeasures that detect or avoid this attack vector or use separate power and clock lines for all tiles (resp. shares).

Safe Error Attack. We point out a specific attack that targets any countermeasure against a probing and faulting adversary. In a safe error attack [186], the attacker perturbs the implementation in a way that the output is only affected if a sensitive variable has a certain value. The attacker learns partial secret information by merely observing whether or not the computation succeeds (*i.e.* does not abort). Consider for example a shared multiplication of a variable x and a secret y and call the resulting product $z = xy$. The adversary faults one of the inputs with an additive nonzero difference such that the multiplication is actually performed on $x' = x + \Delta$ instead of x. Such an additive fault can be achieved by affecting only one share/tile. The multiplication results in the faulty product $z' = z + \Delta \cdot y$. The injected fault has propagated into a difference that depends on sensitive data (y). As a result, the success or failure of any integrity check following this multiplication depends on y. In particular, if nothing happens (all checks pass), the attacker learns that y must be 0.

Among existing countermeasures against combined attacks, none provide protection against this kind of selective failure attack as they cannot detect the initial fault Δ. The attacker can always target the wire running from the last integrity check on x to the multiplication with y. We believe CAPA is currently unique in preventing this type of attack. One can verify that the MAC-tag checking step in a Beaver operation successfully prevents Δ from propagating to the output. This integrity check only passes if all tiles have a correct copy of the public value ε. Any faults injected after this check have a limited impact as the calculation finishes locally. That is, once the correct public values are established between the tiles, the shares of the multiplication output z are calculated locally without further communication among tiles. The adversary is thus unable to elicit a fault that depends on sensitive data.

On the contrary, M&M does not provide protection against Safe Error Attack. In M&M there are no public values as in CAPA, and the multiplication is not performed locally. This means that the result of the multiplication is the actual product $z' = z + \Delta \cdot y$, which avoids further propagation of the fault in case $y = 0$, and hence, resulting in the fault not detected.

PACA. We claim security against the passive and active combined attack (PACA [6]) on masked AES described because CAPA does not output faulty ciphertexts and neither does M&M.

A second attack in this work uses another type of safe errors (or *ineffective faults* as they are called in this work) which are impossible to detect. The attacker fixes a specific wire to the value zero (this requires the n_f-faulting capability) and collects power traces of the executions that succeed. This means the attacker only collects traces of encryptions in which that specific wire/share was already zero. The key is then extracted using $n_{\mathcal{T}} - 1^{\text{th}}$-order SCA on the remaining $n_{\mathcal{T}} - 1$ shares. This safe error attack however falls outside our model since the adversary gets access (either by fault or SCA) to all $n_{\mathcal{T}}$ (resp. $d+1$) shares and thus $(\mathcal{F}_1 \cup \mathcal{P}_1) = \mathcal{T}$.

SIFA. There is also an interesting branch of fault attacks that exploits ineffective faults apart from PACA. A flavor of Ineffective Fault Analysis (IFA) [60] called SIFA [77] was proposed in CHES 2018. For example, a stuck-at-zero fault on a wire or set of wires is *ineffective* when those wires already carry the zero value. This type of faults are naturally undetectable at algorithm level, which makes them immune to both detection and infection countermeasures. SIFA collects a subset of correct ciphertexts from a large number of faulted encryptions and exploits the fact that the intermediate state of the algorithm is not uniformly distributed in this subset. This attack has been extended to masked implementations [75]. Ineffective faults (and thus SIFA) fall outside of our adversary models since they are impossible to detect. Protection against such attack can be provided at a different level, for example using a protocol that erases the key as soon as a certain threshold of faulty ciphertexts has been detected.

5.4.4 On the MAC Tags

What Does Our MAC Security Mean? We stress that CAPA provides significantly higher security than existing approaches against faults. An adversary that injects errors in up to n_f tiles cannot succeed with more than the claimed detection probability. This means that our design can stand $d'_f \gg n_f$ shots if they affect at most n_f tiles. This is the case even if those n_f tiles leak their entire state; hence CAPA's resistance against combined attacks. The underlying reason for this is that to forge values, an attacker needs to know the MAC key, but since this is also shared, the attacker does not gain any information on the MAC key and their best strategy is to insert a random fault, which is detected with probability $1 - 2^{-km}$.

In the case of M&M, the model is effectively weakened for the side-channel adversary, and, at the same time, for the combined adversary, not providing provable security anymore. On the contrary, the security against faults is equivalent to CAPA since the same MAC tags approach is similarly used in this countermeasure.

How Much Do Tags Leak? The tag shares τ_i^a form a Boolean masking of a variable τ^a. This variable τ^a itself is an information theoretic MAC tag of the underlying value a and can be seen as a multiplicative share of a with α. We therefore require the MAC key to change for each execution. Hence MAC tag shares are a Boolean masking of a multiplicative share and are expected to leak very little information in comparison with the value shares themselves.

Forbidding the All-0s MAC Key. The event that the MAC key is zero occurs with probability 2^{-km}. The case of $\alpha = 0$ is a special and dangerous case since the full MAC tag datapth is zeroized, and the fault protection removed. An attacker could inject a fault solely in the datapath, and the tags would be still consistent as they would be still simply zero. Since α is secret and shared, the adversary cannot know when the tags are zero and must still guess α to determine what fault to inject in the tag computation. An adversary strategy of not injecting faults in the tag computation, *i.e.* injecting faults only on the value computations, corresponds to guessing that $\alpha = 0$ and succeeds with probability 2^{-km}. This is completely analogous to guessing for example that $\alpha = 1$ and injecting identical faults in both the tag and value computation accordingly. Either by guessing α or by injecting a random fault, the adversary hits the correct value with probability 2^{-km}, corresponding to our claimed EDP of $1 - 2^{-km}$. Hence, in theory, the case $\alpha = 0$ is equivalent to any other nonzero MAC key. In practice however, the strategy corresponding to guessing $\alpha = 0$ is easier since it requires only fault injections in the value datapath and not the tag datapath. To avoid it, one could exclude the zero MAC key and reduce the EDP to $1 - (2^{km} - 1)^{-1}$. This difference is negligible if km is sufficiently large. Additionally, if the MAC key size mk is small, we should forbid the all-0 MAC key. This ensures that tags are injective: if an attacker changes a value share, he must change the tag share. Note that for M&M, in the case of forbidding all-0 MAC key, the infective computation output phase can no longer be used, since it requires α to be uniformly random in $\mathrm{GF}(2^{km})$.

5.4.5 Differences between CAPA and SPDZ

Offline Phase. In SPDZ, the auxiliary data is generated using a somewhat homomorphic encryption scheme. The mapping onto a chip environment thus seems prohibitive due to the need for this expensive public-key machinery to obtain full threshold and the large storage required. We avoid this by generating the Beaver triples using passively secure shared multipliers. Furthermore, to avoid the large storage requirement, we produce the auxiliary data on the fly whenever required.

MAC Tag Checking. SPDZ delays the tag checking of public values until the very end of the encryption by using commitments. For this, each party keeps track of publicly opened values. This is to avoid a slowdown of the computation and because in the MPC setting, local memory is cheaper than communication costs. In an embedded scenario the situation is opposite so we check the opened values on the fly at the cost of additional dedicated circuit. In hardware, we "simulate" the broadcast channel by wiring between all tiles. Each tile keeps a local copy of those broadcasted values.

Adversary. Although MPC considers mainly the "synchronous" communication model, the SPDZ adversary model also includes the so-called "rushing adversary". In our embedded setting, as already pointed out, the "rushing" adversary is impossible since the communication is considered atomic and synchronous. Due to the nature of the implementation, the computational environment and storage are highly constrained. On the other hand, communication channels are very efficient and can be assumed to be automatically synchronous with all tiles progressing in-step in the computation.

5.5 Conclusion

In this chapter, we have introduced the first adversary model that jointly considers side-channels and faults in a unified and formal way. The *tile-probe-and-fault* security model extends the more traditional wire-probe model and accounts for a more comprehensive adversarial behavior. Within this model, we developed the methodology CAPA: a new combined countermeasure against physical attacks. CAPA provides security against higher-order DPA, multiple-shot DFA, and combined attacks. CAPA scales to arbitrary security orders and borrows concepts from SPDZ, an MPC protocol. We have also shown the

feasibility of implementing CAPA in embedded hardware by providing prototype implementations of established block ciphers.

Then, we have introduced a new family of countermeasures to provide security against both SCA and DFA. M&M can extend any masking countermeasure with information-theoretic MAC tags and infective computation. We have demonstrated how to construct basic M&M building blocks and how to build a secure implementation of any cipher. We have illustrated our proposal with first- and second-order secure implementations of AES, and we have experimentally verified the SCA and DFA security. We have shown that M&M implementations can be very efficient while providing resistance against both SCA and DFA in a robust but realistic adversary model.

Finally, we have presented a security analysis against SCA, FA, and combined attacks for both countermeasures. We have provided further analysis on why the presented schemes provide (or not) protection against specific fault and combined attacks from the literature. Moreover, we have included a discussion on the MACs themselves, and how CAPA differs from the MPC scheme SPDZ.

Chapter 6

Verification of SCA and FA countermeasures - VerMFi

In this chapter, we introduce VerMFi: Verification tool for Masked implementations and Fault injection. It is a set of open-source verification tools to aid during the design flow towards secure hardware implementations for both side-channel and fault analysis. The resulting tool is the composition of two works, VerMI (Verification tool for Masked Implementations), and VerFI (Verification tool for Fault Injection). Fig. 6.1 presents the tool's flow diagram.

VerMI takes care of the Side-Channel Analysis evaluation, helping to spot any mistake with bit precision when masking a cryptographic hardware implementation. It provides an initial univariate security evaluation without any special equipment. It works at both algorithmic and implementation levels, with an inherent capability to check designs in the presence of glitches by checking the properties defined in Threshold Implementations. The tool was tested with different implementations from the literature, successfully finding a flaw in the round-based Keccak implementation of [100].

VerFI takes care of the Fault Analysis evaluation, allowing the user to rapidly examine the fault detection/correction capabilities of a given implementation in the desired fault model. We introduce the concept of Cryptographic Fault Diagnosis, which revises and shapes the notions of fault diagnosis in reliability testing to the needs of evaluating cryptographic implementations. VerFI materializes the idea of Cryptographic Fault Diagnosis. It is a fully automated, fault detection tool, processing the gate-level representation of arbitrary cryptographic implementations. Among several case studies, we

demonstrate its application on an implementation of the LED cipher protected with ParTI [168], a combined countermeasure against side-channel analysis and fault-injection attacks published at CRYPTO 2016. This experiment revealed general implementation flaws and undetectable faults leading to successful DFA on the protected design with full-key recovery.

Through the chapter, we first revisit previous work and state of the art regarding automated verification tools for both side-channel and fault analysis. Second, we introduce the logic and fault simulators from which VerMFi builds upon. Then, we present the features of both VerMI and VerFI, and how they are implemented, followed by several case studies and relevant findings from the experiments with both tools. Finally, we report the results of the previous analysis and several performance metrics.

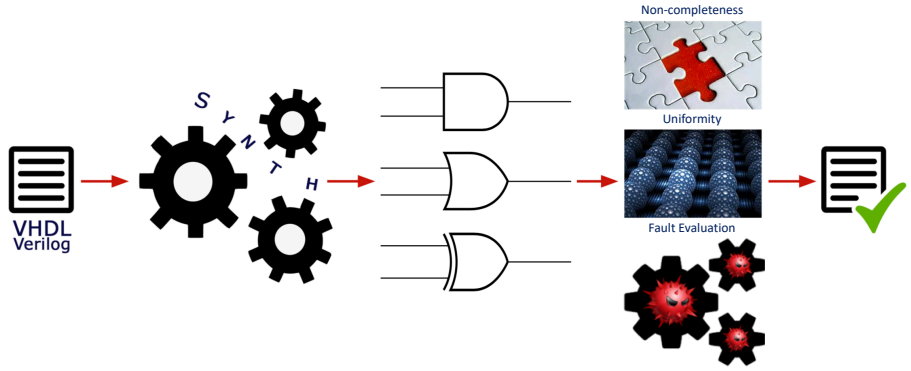


Figure 6.1: VerMFi's flow diagram.

This chapter is based on two works, published in International Conference on Electronics, Circuits, and Systems (ICECS) 2018 [12], and International Symposium on Hardware Oriented Security and Trust (HOST) 2020 (1) [14], which were carried in collaboration with Svetla Nikova, Vincent Rijmen, Felix Wegener, and Amir Moradi. I am the main contributor of the work, except for the DFA on ParTI described in § 6.4.4, which was done together with Felix Wegener.

Previous work

Formal verification for masked implementations is a widely researched topic, which started with multiple works for analyzing software implementations [18, 23,68,81,82,142]. Since these papers do not take into account physical non-linear effects produced by the hardware, *e.g.*, glitches, they can not be applied to

verify hardware implementations. Recently, several manuscripts have appeared proposing verification methodologies for hardware implementations [17, 36, 43, 173].

Formal verification of masked hardware implementations can be done at different stages of the design flow: the algorithmic, the implementation, or the physical level. In the work by Reparaz [156], a tool for verification at the algorithmic level is presented. The tool receives a software implementation of the secured function and performs a leakage assessment over simulated traces. However, glitches are not taken into account. Between the algorithmic and the implementation levels, we can position `maskVerif`, from Barthe *et al.* [17]. They present a tool to generally evaluate the security of higher-order masking within the ISW or probing model, with and without glitches. Additionally, they can assess whether the implementation is NI or SNI [18]. This tool expands from the one in [18], improving performance, and considering also glitches.

At the implementation level, Bertoni *et al.* [36] propose a tool suitable for hardware implementations in the presence of glitches. Nevertheless, their tool only analyzes combinational logic with a simple power model. Bloem *et al.* [43] also introduce a formal verification method for hardware implementations in the presence of glitches. Given a hardware implementation, they extract the netlist and model the logic gates using a Fourier expansion. Three different types of variables are defined, *i.e.*, secret variables (S), masks (M), or other variables (P). Every intermediate value of the circuit is *labeled* with a set of these variables, where the labels are based on the Fourier representation of the Boolean function. A value is not secure if the associated label contains a set with only secret variables. Glitches are simulated by extending the representation of a gate to compute any Boolean function from its original inputs. Additionally, to analyze higher-orders, a SAT solver is instantiated.

Also at the implementation level, there is the tool Computer Aided Side Channel Analysis Design Environment (CASCADE), by Šijačić *et al.* [173], which evaluates the security of a hardware implementation at an advanced stage of the design flow. Based on the parasitic effects of the circuit, and using different models for the power consumption of the circuit, power traces are built using Spice. These simulated power traces are processed using the Test Vector Leakage Assessment (TVLA) methodology to find leakage.

Finally, at the physical level, TVLA [66] is widely utilized to address the leakage of implementations (see § 2.8), or attack based evaluations, which consist of launching several side-channel attacks. One of the downsides of these methods is that ASIC or FPGA implementations are needed to perform these evaluations, apart from additional equipment for the experiment. Furthermore, in the case of TVLA, the conventional non-specific test does not provide information about

the exploitability of the leakage in case it is spotted. Moreover, in the case of attack based evaluation, it only guarantees resistance against the executed attacks.

While many tools for verification of side-channel resistant implementations exist, ranging from purely formal verification to more practical evaluations, no automated procedure has yet been suggested in the literature to evaluate the countermeasures against fault-injection attacks. Several previous tools can assist the automated fault analysis of a formal cipher description to identify and facilitate potential fault attacks. Zhang *et al.* [188] suggest a method to automate Algebraic Faults Attacks (AFA) using SMT solvers, while more recent papers of Khana *et al.* [120] and Saha *et al.* [164] propose frameworks for fault characterization in cryptographic algorithms tailored to Differential Fault Analysis (DFA) attacks. We can divide the most recent procedure [164] into three phases: identification of essential distinguishers, data-mining of fault simulation data, and finally, evaluation of the attack complexity. These tools can surely facilitate the emulation of an active attack. However, assuming a countermeasure to fault-injection attacks where the adversary's boundary is defined, no publicly available tools are currently known to (automatically) examine the security of such an implementation under the defined adversary model.

Apart from the difficulty of practical evaluations, it is highly challenging to limit the injected faults based on the assumed fault model, e.g., when injected by clock glitch. Multiple commercial Electronic Design Automation (EDA) tools feature powerful fault simulators, tailored to address IC testability. The typical workflow of these tools is to generate a scan circuit by replacing the functional flip-flops with scan flip-flops [1], and then run an Automatic Test Pattern Generation (ATPG) tool which only handles the combinational part of the circuit provided the full scan design. There are also several manuscripts on open-source fault simulators for fault diagnosis. The first one, known as PROOFS [146], implements a single stuck-at fault simulator. Follow-up study [125], optimizes previous fault simulator without changes in the fault model. Later, the simulator [46] extends this to multiple and single stuck-at faults, and a subset of Single Event Upset (SEU) limited to registers, which hampers fault injection inside combinatorial circuits. In addition to this, it is not possible to specify when the fault should be injected, *i.e.*, at which clock cycle(s).

[1] A scan flip-flop consists of a multiplexer deriving the input of an ordinary flip-flop. This enables forcing any arbitrary value to the flip-flops.

6.1 Logic and Fault Simulators

In this section, we introduce the underlying logic and fault simulators that implement the necessary procedures to later evaluate the security of a design. We start by presenting the necessary objects which we use to build a structural model of the circuit, followed by the simulation procedure used to evaluate it. Finally, we elaborate on how faults are modeled, the different injection mechanisms, the fault simulation, and the input test vectors to use to perform the simulation.

6.1.1 Logic Simulator

Our tool receives as input an HDL file that may be written in VHDL or Verilog. To unify the structure of the files, we first synthesize the HDL using RTL synthesis tools with open source libraries to extract the netlist. The tool creates a specific *.tcl* file for the design to analyze, automating the RTL synthesis as well as the netlist generation. The synthesis includes several necessary constraints to prevent unwanted optimizations, which could harm the security of the analyzed design. The tool parses the netlist, and a structural model of the circuit in the shape of a Binary Decision Diagram (BDD) is built using wires, pins, and gates as objects connected among them. We create three classes that allow us to build the structural model of a given circuit. The objects declared correspond to the essential logic elements:

Wire. A wire needs to hold name and value. This first class includes several attributes: the name of the wire, the value, and a flag to note whether the wire is already evaluated, among others. It includes the method `eval_wire()` that evaluates the value of the wire, sets the flag to "evaluated," and propagates the simulation to the next gate.

Gate. A gate requires, primarily, a type and functionality. This class keeps the gate type (AND, XOR, MUX), the name of the block instantiation, and the functionality, a function that evaluates the output according to the type of gate. We implement an abstract basis class that contains the attributes and methods common to every gate. We then create multiple derived classes, one for each different type of gate. Each class implements the specific functionality of the element and includes specific attributes if needed.

Pin. The third class is created to connect the previous two. The presence of this union is crucial since it determines the role of each wire when evaluating a gate. It stores the name of the pin, used later when computing a gate function, and the wire and the gate to which the pin is connected.

Connecting elements. For each wire, gate, and pin, the tool instantiates a new pointer to the specific object. The class `wire` has two pointers to pin, `pinFrom_`, and `pinTo_` that define the beginning and the end of the wire. The class `gate` holds a vector of input pins `inputs_` and another pin `output_` for the output. Pins are also essential to model fan-outs. In digital logic, fan-outs occur when a logic gate output is fed into multiple subsequent gates as input. We model them by adding two more attributes to the wire class: the first one, a flag to indicate whether the wire has a fan-out, and the second one, a vector of pins to keep the multiple branches of this wire, `fanOutPins_`. These new pins represent the fan-out wires that are driven to the different gates.

We implement a factory method `GatesFactory`, in charge of creating every gate according to the type, and making all the connections in order to build the structural model.

With this structural representation, we implement an *event-driven* simulator. We model the sequential behavior of the circuit, and the concept of time provided by the clock with a simple scheduler in the form of a while-loop, where each iteration corresponds to a clock cycle. The simulation finishes when a particular *done*, or *ready* signal (as a primary output of the circuit) is set to '1/0', or when the loop exceeds the number of iterations requested by the user. We use a unit-delay timing model, where all gates and interconnects have the same delay, ignoring transient values since we only consider '0' or '1' values. We model the time to advance in discrete "jumps", where this notion is created by the event-driven simulation and the order of execution of the events. Our simulator is compatible with either round-based and serial implementations, or merely isolated pipelines.

6.1.2 Fault Simulator

Hereunder, we describe the main parts of the fault simulator, which include how faults are implemented, the fault injection mechanisms that allow the user to emulate any fault model chosen by the user, and how the simulation is performed.

Our fault simulator extends the faults covered in [46], accepting SEUs in every component of the circuit and allowing the user to specify when precisely the

fault(s) should be injected, to fully comply with the possible fault models defined by the user.

Fault. We define a new class `fault`, which has three attributes: active, cycle, and type. These three attributes respectively define whether the fault should be injected or not, at which cycle, and the type of the fault.

A `fault` object is appended to every gate (comb. or seq.) and triggered when the cell is evaluated. It modifies the gate output (if active) according to the fault type and the cycle. Several fault attacks need a fault to be injected at the input of a non-linear function of the cipher. Hence, we insert buffer gates at the inputs of hierarchical sub-modules with a `fault` appended as well. These buffers enable the possibility of injecting faults at such input signals.

Injection Mechanisms

The fault injection mechanism is the most delicate feature as it determines the fault model used for the evaluation. It has to be flexible, adaptable to as many scenarios as possible, allowing the users to test their designs with the fault model of their choice.

The user has two fault configuration files available in which the appropriate options can be specified. Each file defines a different kind of fault injection mechanism, either *hierarchical* or *component-wise*. In both of them, the user defines the number of fault simulations N_s that the tool has to perform, and the number of faults N_f to inject per simulation. In the hierarchical injection, the user should additionally define the maximum number of faults per clock cycle N_t. For every module, the user should decide whether or not to enable fault injection, whether to activate the gate-input model, when to inject the fault, and which kind of (SEU or SA0/1).

Hierarchical Injection. The first file drafts the hierarchical structure of the design with the respective module names, from which the tool stores all the cells from the modules enabled to be faulty and in which `cycle(s)`. A set of faults is created as $\boldsymbol{\mathcal{F}} = \{f_1, \ldots, f_{N_e}\}$, where f_i is the identifier of the ith cell enabled to be faulty, and N_e the total number of such cells. The tool selects up to N_f random faults of $\boldsymbol{\mathcal{F}}$ per injection $\mathcal{I}$ and repeats this for up to N_s times. The user can perfectly delimit the areas where the faults should be injected, with equal probability.

Component-wise Injection. The second file details every module's gates and (optionally) its inputs, allowing the user to precisely decide where to fault. This functionality is meant to test very particular faults, usually to be used after a more generic test with the hierarchical injection, ideal for launching an actual attack, or just debugging. Moreover, this injection mechanism allows exhaustive fault evaluation, provided that the total number of faults is limited. Similarly to the previous modality, the tool collects and stores all faults into $\boldsymbol{\mathcal{F}}$.

On the one hand, if $N_f = N_e$, the tool performs a single simulation activating all such faults. On the other hand, if $N_e > N_f$, then the tool simulates all possible combinations of N_f faults in the set $\boldsymbol{\mathcal{F}}$. If they exceed a user-defined threshold, the injection is done at random performing a max. of N_s fault simulations. Otherwise, N_s is ignored.

Word-wise Injection. The user can also decide to inject the faults in a word-wise fashion, *e.g.*, byte- or nibble-wise faults. The idea is to enable fault injection into the components driving all wires of the targeted bus. In this case, the tool can inject a random offset Δ into the targeted bus (SEU fault model), or randomly set/reset some bits of the selected word (Stuck-at-0 (SA0) and Stuck-at-1 (SA1) fault models). Depending on how the hierarchical design of the circuit, it would suffice to enable the `fault` flag of the blocks treating such buses, or, otherwise, make a more careful selection with the component-wise injection.

Fault Simulation

Faults are processed at the right iteration (cycle), activated, or deactivated according to the specified clock cycle(s), modifying the output of an evaluated model accordingly.

The structure of our fault simulator is similar to the one described in Algorithm 2 of [46]. In order to perform the simulation, the sequential circuit is "opened" by removing the registers, providing the circuit with new primary inputs (taken as the output of the register), and primary outputs (provided by the input of the register). Together with the original inputs, the registers' values are fed into the combinational circuit. When the current cycle simulation is over, registers are updated with the new value, which will be primary inputs in the next cycle.

Our goal is not to design the most efficient general-purpose fault simulator, but to design a robust fault evaluation framework tailored to cryptographic implementations, which blends the concepts of fault diagnosis and fault evaluation in cryptographic algorithms. Thus, we start with a serial fault

simulator, which simulates one by one the given fault injections, *i.e.*, one injection per simulation.

Input Test Vectors

The circuit input stimuli is another variable that influences the total space of possible fault simulations, together with the fault location and the time. The impact of different inputs on the coverage is considerable since, for example, a single-bit fault could be classified as non-detected for a particular input, while for different input, the fault might not propagate and thus classified as ineffective. Hence, in order to have a meaningful result, the more inputs provided to the evaluation, the higher the confidence. Since it is impractical to evaluate all possible inputs exhaustively, VerFI gives the user two options to specify the inputs for the experiment:

- The user can specify the number of different inputs for simulation, and the tool would choose them at random.
- The user can provide a file with a list of input test vectors. For every input to simulate, the tool performs the same experiment, *i.e.*, the same fault injections.

One could argue that choosing the input test vectors smartly and efficiently determines the evaluation result. On the one hand, the user can do that by providing the inputs for the simulation himself. On the other hand, in traditional reliability testing, the generation of random test vectors can already detect up to 90% of the faults approximately. Only for the most challenging faults advanced ATPG techniques are needed to find test vectors to detect them. With this argument in mind, and taking into account our personal experience from multiple experiments, if a device were to be flawed, feeding random inputs suffices to spot the vulnerabilities.

6.2 VerMI

VerMI practically evaluates the TI properties (excluding correctness) to ensure the correct application of a given masking scheme. It offers two main functionalities: checking non-completeness and checking uniformity in both combinational and sequential circuits.

6.2.1 Header

The top module of the design to analyze must include a small header in which the user specifies essential data for the tool. The header includes the names of the sensitive data inputs, the outputs of every layer of registers, the randomness, and the outputs to be analyzed. Thus, the tool can distinguish sensitive data (protected data) from public inputs and every stage of registers, random data, and sensitive data outputs. Variables and shares are sorted in an N by M matrix, where N is the number of variables and M the number of shares (Fig. 6.2a). This data structure will help the tool to evaluate Non-completeness.

6.2.2 Non-completeness

VerMI can check any-order non-completeness and output the specific bits that fail to satisfy this property.

Input dependencies. We first collect the input dependencies of every primary output with a recursive search through the BDD. The algorithm starts at a primary output and goes through the circuit backwards, covering all paths containing this output. The algorithm keeps running until it arrives at a primary input, which is saved as a dependency if it is declared as sensitive in the header.

Decision algorithm. Then, we check whether all shares for the same variable are included. If they are all included, the variable fails non-completeness. The tool goes through every output performing this check, reporting the ones that fail.

To check whether all shares for the same variable are included is not a trivial task since the tool is unable to distinguish between variables and shares. The tool constructs the same sensitive data matrix as described in 6.2.1, with the input dependencies of every output. Then, it checks whether any variable uses all shares by checking whether there is any full row in the matrix. Fig. 6.2 presents the tables for both third output shares of Example 6.1, as well as the full sensitive data matrix. Example 6.1 presents a non-complete and a complete

TI sharing of $z = a \oplus bc$, with $(d, t, s_{in}, s_{out}) = (1, 1, 3, 3)$:

Example 1

$$z_1 = a_1 \oplus b_1c_1 \oplus b_1c_2 \oplus b_2c_1 \qquad z_1 = a_1 \oplus b_1c_1 \oplus b_1c_2 \oplus b_3c_1$$

$$z_2 = a_2 \oplus b_2c_2 \oplus b_2c_3 \oplus b_3c_2 \qquad z_2 = a_2 \oplus b_2c_2 \oplus b_2c_3 \oplus b_3c_2$$

$$z_3 = a_3 \oplus b_3c_3 \oplus b_1c_3 \oplus b_3c_1 \qquad z_3 = a_3 \oplus b_3c_3 \oplus b_1c_3 \oplus b_2c_1$$

Non-complete *Complete*

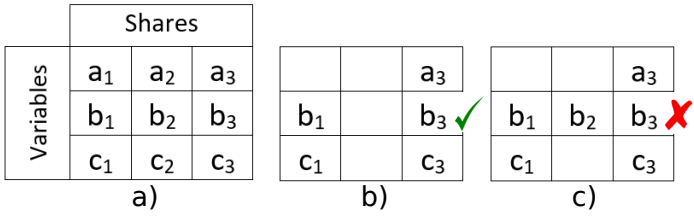


Figure 6.2: Left to right: complete sensitive data set, third share (Ex. 6.1) non-complete, and third share (Ex. 6.1) complete.

Single bit variables. The tool handles variables from the analyzed HDL module bitwise. Managing an entire bus, for example, would be faster, but also less detailed. This distinction is crucial to make the tool able to catch all possible dependencies and thus to have a robust non-completeness analysis.

d^{th}-order non-completeness. All possible combinations of d outputs are analyzed with the same method presented above. In this case, the dependencies of the d outputs are combined, building the data matrix with all of them. All combinations of d output variables that together fail non-completeness are reported. Currently, VerMI supports up to third-order non-completeness verification. However, it can be extended to support the verification of any meaningful order.

Dealing with glitches. The structural model built by VerMI ignores the timing of the gates. How, then, are glitches detected? We do not track glitch dependencies themselves; instead, we check for non-completeness. Hence, if this property is fulfilled, it ensures security in the presence of glitches [147].

6.2.3 Uniformity

The following functionality is to check whether uniformity is preserved across operations. The tool calculates the output distribution of a given module by simulating all possible inputs. Then, it calculates the frequencies of occurrence of each output vector for the different input sharings. If all frequencies are the same inside every group of input sharings, then the module is said to *preserve uniformity*. To generate uniformity-preserving shares is easy for invertible linear operations, but not for non-linear operations, since the interaction between shares is needed. This functionality is meant to analyze these functions separately and check whether they preserve uniformity.

Evaluation. We use the formulas from [40] to evaluate the uniformity. We check whether, for a particular unshared input, every output has the same frequency of occurrence when all possible sharings of this input are evaluated. The same procedure is repeated for every possible unshared input together with, if added, all random bits specified. An illustration of this method can be found in Fig. 6.3, where the uniformity of a 3-share AND gate without randomness is analyzed (See Eqn. (4.1)).

x	y	$z_1z_2z_3$							
		000	011	110	101	001	010	100	111
0	0	7	3	3	3	0	0	0	0
0	1	7	3	3	3	0	0	0	0
1	0	7	3	3	3	0	0	0	0
1	1	0	0	0	0	5	5	5	1

Figure 6.3: Uniformity check example.

Limitations. It is important to note that the uniformity check is limited by the total number of input bits and random bits. As explained above, the program has to evaluate all possible inputs to the circuit and process the output. Naturally, it is only feasible to check modules with a small number of inputs, for instance, the non-linear operations (S-boxes) in a cryptographic algorithm.

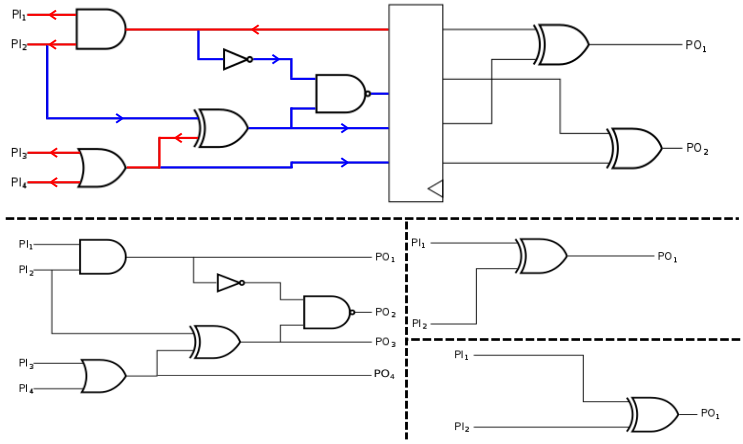


Figure 6.4: Top: sequential circuit. Bottom: combinational sub-circuits for first and second stages.

The designer needs to provide to the tool the sub-module that implements the S-box.

6.2.4 Sequential circuits

In this section, we present how VerMI handles sequential circuits. The analyzed circuit is divided into all possible combinational sub-circuits to evaluate non-completeness, or simulated altogether to analyze uniformity.

When designing for security, apart from functionality, the registers are required to prevent the propagation of glitch dependencies and to synchronize refreshing. Uniformity is not affected by the presence of registers. Thus, the registers' primary goal, security-wise, is to split the design into different sub-circuits to prevent glitch propagation. Therefore, the tool analyzes non-completeness over the combinational logic in between registers, while uniformity is analyzed with registers included.

For a tool, it is not trivial to divide a netlist into different sub-circuits. It is imperative to include the specific components that correspond to a certain sub-circuit in order to perform accurate security analysis. Missing an element or including extra components, would result in a possibly non-complete output that actually should have failed the test, or, on the other hand, in extra dependencies that are not in the circuit, producing a false positive. We design a *Back-and-*

forth recursive algorithm to split the netlist correctly. Fig. 6.4 illustrates how the algorithm works.

***Back-and-forth* algorithm.** To do this "physical" separation in the simulated circuit, we design the *Back-and-forth* algorithm and a new class `Subcircuit` that includes all inputs, outputs, wires, and gates of a sub-circuit. The algorithm works as follows:

1. All primary outputs and the input of every flip-flop (which are now new primary outputs) are stored in a new vector. The same is done with all primary inputs, including flip-flop outputs.
2. Create a new `subcircuit`, pick one of the outputs of this new vector, and remove it.
3. Perform a *backward* tree search on this output (depicted in red and with a left arrow in Fig. 6.4) to find the corresponding primary inputs. The algorithm will keep every gate found in its passage through the circuit.
4. When the algorithm reaches a primary input, it is stored (if it was not already), and then it calls a function to perform another tree search *forward* (blue and right arrow) to find all the outputs that might depend on this input. When an output is reached, it is stored (if it was not already) and erased from the new outputs vector. If there were no fan-outs in the sub-circuit, no new outputs would be stored.
5. Pick a new output from the new outputs vector, and go back to 3 with this new output.
6. When no new outputs are found, the recursion finishes and the sub-circuits are complete. Go back to 2 until the new outputs vector is empty.

Note that this algorithm may create independent sub-circuits from the same stage of a pipeline, even though visually, they could be considered to belong to the same sub-circuit. This division happens when there is not a fan-out that connects them. In Fig. 6.4, the right stage sub-circuit would be divided in two by the tool, as it should be, since there is no connection whatsoever between the two of them. When analyzing higher-order non-completeness, independent sub-circuits from the same stage are analyzed together. Note also that in the case that a full circuit is analyzed, including control logic, for instance, there exists the possibility to partition the circuit into only one sub-circuit. This sub-circuit is essentially the original circuit opened by the registers, with new primary inputs and outputs created. Thus, the sensitive data associated with every layer is associated instead with every primary output.

Algorithm 2 presents the pseudo-code for the above-presented algorithm. We use the notation ${}_{w\rightarrow}g_{\rightarrow v}$ to refer to a gate g whose output is v and has an input w. Similarly, we use ${}_{w\rightarrow g\rightarrow}v$ to refer to the output v and $w_{\rightarrow g\rightarrow v}$ for the input w.

Simulation. The simulation of registers is only needed for uniformity analysis, which is not affected by the fact that registers are synchronization elements. Thus, the simulator ignores the concept of time, and registers are treated as plain buffers, speeding up greatly the analysis of uniformity.

Algorithm 2 Recursive *Back-and-forth* algorithm

```
1: procedure SUBCIRCUIT_SEARCH_BACK(wire w, subcircuit subcirc, wires remaining_outs)
2:     if w is a Primary Input or w is a Flip-Flop output then
3:         if w is not yet saved then
4:             subcirc.inputs ← w
5:             SUBCIRCUIT_SEARCH_FORTH(w, subcirc, remaining_outs)
6:     else
7:         for all inputs i in g do
8:             SUBCIRCUIT_SEARCH_BACK(i, subcirc, remaining_outs)
9:         end for
10: end procedure

11: procedure SUBCIRCUIT_SEARCH_FORTH(wire w, subcircuit subcirc, wires remaining_outs)
12:     if w is a Primary Output or w is a Flip-Flop input then
13:         if w is not yet saved then
14:             subcirc.outputs ← w
15:             Erase w from remaining_outs
16:             SUBCIRCUIT_SEARCH_BACK(w, subcirc, remaining_outs)
17:         end if
18:         if gt_{→w} is not yet saved then subcirc.gates ← gt_{→w}
19:         if w belongs to a fan-out then
20:             for all wires fo emerging from w do
21:                 SUBCIRCUIT_SEARCH_FORTH(fo, subcirc, remaining_outs)
22:             end for
23:         end if
24:     else
25:         if _{w→}g is not yet saved then
26:             subcirc.gates ←_{w→} g
27:         end if
28:         SUBCIRCUIT_SEARCH_FORTH(_{w→g→}v, subcirc, remaining_outs)
29:         if w belongs to a fan-out then
30:             for all wires fo emerging from w do
31:                 SUBCIRCUIT_SEARCH_FORTH(fo, subcirc, remaining_outs)
32:             end for
33:         end if
34:     end if
35: end procedure
```

6.3 VerFI

In this section, we present the details of our Verification Tool for Fault Injection (VerFI). We first introduce the concept of Cryptographic Fault Diagnosis (CFD), followed by VerFi's framework and its functionalities to emulate the CFD concepts successfully. Then, we explain how the tool evaluates a countermeasure. Finally, we explain how the tool can be used to address SIFA resistance and to evaluate infective countermeasures.

6.3.1 Cryptographic Fault Diagnosis

We aim to bring the fault diagnosis knowledge from reliability testing to the cryptographic setting for the evaluation of fault countermeasures. We adapt these concepts to the needs of cryptography to define the notion of Cryptographic Fault Diagnosis. We compare the goal of testing in IC design versus the goal of evaluating fault-resistant implementations of cryptographic algorithms. Furthermore, we introduce a new fault model to complement the current state of the art and provide a consolidated way of characterizing any fault model.

Conventional vs. Cryptographic Fault Diagnosis

The process of reliability testing and evaluation of fault-resistant cryptographic implementations differ in multiple concepts:

- The **primary goal** is the principal difference between both procedures. While the first one aims at ascertaining whether a defect occurred during the manufacturing process, the second one's goal is to evaluate the performance of a countermeasure against fault-injection attacks.
- **Detecting** a fault in reliability testing means that the fault propagates to the outputs of the circuit, changing their value with respect to the fault free outputs. On the other hand, in cryptography, a fault is detected when the integrated countermeasure reacts to a faulty execution, *e.g.*, by raising an abort or status signal, or by propagating an error randomizing the internal state.
- The **origin** of faults in the first scenario comes from small defects during the manufacturing process, which makes these faults purely random and unpredictable. On the contrary, cryptographic implementations face malicious users injecting intentional faults, specially crafted to bypass the security of the system.

- The **number** of simulated faults also differs. Circuits are tested with single-bit faults due to the high coverage this model provides and the impossibility of simulating all combinations of multiple faults. Conversely, the methods used to inject malicious faults often target multiple faults at the same time, *e.g.*, laser fault injection.
- **Coverage** in reliability testing (associated with a test set) is defined as the ratio of faults that the stimuli test set can detect over the total number of possible faults in the circuit. The coverage in a cryptographic implementation is associated with the underlying countermeasure. It is defined as the ratio of faults detected over the total number of faults injected.

Given the differences mentioned above, a simulator with a different testing approach is needed to conduct a successful cryptographic fault diagnosis using particular fault simulation methodologies. The *result* of the test shall give an accurate assessment of the performance of the countermeasure. The simulator needs to *identify* the detection mechanism and to recognize whether the circuit reacts to the injected fault. Probably as the most challenging task, the simulator has to be highly flexible to be able to emulate all different possible *origins* (fault model defined by the user), which represent the hypothetical attack methodology. Due to the need to simulate *multiple*-fault injections, the use of deductive or concurrent fault simulation becomes impractical when considering a high number of possible combinations. Hence, we identify more suitable methodologies, such as serial or parallel fault simulation. Additionally, the current testing procedure commonly used, *i.e.* scan design plus ATPG on the combinational logic, is not suitable for evaluating fault countermeasures since, for example, a checking mechanism could take more than one cycle, and hence, it would need the full sequential simulation. Moreover, the fact that these tools focus on a single fault analysis severely limit their applicability for cryptographic fault analysis. As a final result, meaningful *coverage* has to be reported following the type of implemented countermeasure.

Consolidating Fault Models

The most extensively used model in fault diagnosis for digital circuits is the stuck-at-0/1 fault model. These are faults that occur in the interconnections of Boolean gates, *i.e.*, wires or nets. Additionally, bit-flip faults (SEUs) occurring in the memory elements, among others, are also used.

To define a unified fault model for cryptographic implementations, we benefit from the extensive theory on how faults are modeled in reliability testing,

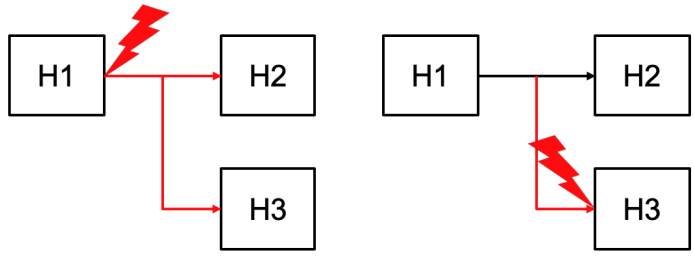


Figure 6.5: Injecting fault on a hierarchical module (H3) from the inputs in the gate-output fault model $\mathcal{M}_{\mathcal{O}}$ (left) and in the gate-input fault model $\mathcal{M}_{\mathcal{I}}$ (right).

in addition to the fault models already presented in the Fault Analysis literature [5, 16, 107, 114, 150]. To encapsulate all possible fault models, we propose a characterization given at four different levels, going from more general notions to narrower properties of the model. The proposed fault model is structured as follows:

Gate-input & -output fault models. We model faults to occur within the Boolean components (gates), rather than on the wires since most of the malicious attacks result in a faulty operation on these components. Given the great variety of fault attacks, with very different effects and accuracy, we model faults in Boolean components as gate-output faults ($\mathcal{M}_{\mathcal{O}}$) where faults occur solely on the output wires, or as gate-input faults ($\mathcal{M}_{\mathcal{I}}$) where faults may occur on any input and the output. The gate-input fault model (analogous to the wire model used in IC testing) is a stronger model due to the finer granularity in the injection. The crucial difference between the two fault models is that with $\mathcal{M}_{\mathcal{I}}$, the attacker is given the ability to hit particular branches of fan-outs going into different components, directly affecting their inputs. On the other hand, in $\mathcal{M}_{\mathcal{O}}$, the gate driving the stem of such fan-out should be targeted, propagating the fault to all fan-out branches. Fig. 6.5 illustrates this concept. Note that, in Fig. 6.5, one could affect the second branch of the fan-out in the $\mathcal{M}_{\mathcal{O}}$ by injecting a fault in H3, but this might not have the same effect as having the fault directly in the inputs.

Conventional fault injection mechanisms used against cryptographic devices, *e.g.*, clock or voltage glitches, are best modeled as gate-output faults. Furthermore, most of the scenarios consider the faults to happen solely at the output of the gates.

Nevertheless, more advanced techniques could be used to inject more precise

faults, and thus, they are best modeled as gate-input faults. These techniques include Electromigration [42], EM glitch, or very accurate laser beams (laser cutter), which can affect input wires of a gate [155]. With highly accurate tools such as Scanning Electron Microscope (SEM) or Focused Ion Beam (FIB), the attacker might be able to inject intra-gate faults, getting a CMOS network to *stuck-at open/close* state. As reported in [87, 88], the most accurate way of modeling these faults is gate-input faults.

Uniform & Biased fault models. We represent a uniform fault model as $\mathcal{M}_{\mathcal{U}}$ and a biased fault model as $\mathcal{M}_{\mathcal{B}}$. In the case of cryptographic implementations, $\mathcal{M}_{\mathcal{U}}$ is less often utilized. The faults are maliciously injected by an attacker, who targets the (most) vulnerable parts of the design, *e.g.*, utilizing a localized laser beam.

Univariate & Multivariate models and number of faults. A univariate fault model limiting the number of faults to t in a single clock cycle is denoted by $\mathcal{M}_t$. A multivariate fault model allowing at most t number of faults per clock cycle is represented as $\mathcal{M}_t^*$.

Type. The type of faults are typically SEU or SA0/1:

- **Single Event Upset (SEU)** is a non-permanent error that affects memory elements (registers) in a digital circuit, causing a bit to flip. Single Event Transient (SET) faults are similar errors that affect the combinational logic, which could turn into an SEU if propagated until a sequential logic (memory) unit. We refer to a non-propagated SET as an *ineffective* fault. For the sake of clarity, throughout the remaining of this work, we treat every bit-flip as a SEU. This type of fault is the one that most realistically models the effect of DFA [37, 44].
- **Stuck-at-0 (SA0)** and **Stuck-at-1 (SA1)** are the most common type of faults used in fault modeling theory. They represent a wire stuck to ground or V_{dd}, respectively. Contrary to this, in the cryptography settings, these kinds of faults are only used to model different fault attacks, like Safe-Error Attacks (SEAs) [186], or Statistical Ineffective Fault Attack (SIFA) [76].

We represent the combination of several of the previous fault models with the symbol $\cap$. That is, $\mathcal{M}_{\mathcal{O}} \cap \mathcal{M}_{\mathcal{B}} \cap \mathcal{M}_3^*$, means we are working on the gate-output and biased fault models, where up to three fault at any clock cycle (multivariate) are allowed.

6.3.2 Framework

VerFI is designed to analyze the coverage of implementations of fault-injection countermeasures at the gate level, achieving an early assessment in the design flow. The execution of the program takes place in two steps, starting with a preprocessing of the RTL files to continue with the actual evaluation of the design.

Preprocessing. In the first stage, the tool produces the fault configuration files for both hierarchical and component-wise fault injection. These files allow the user to define a personalized fault injection. VerFI can additionally work with Yosys [182] synthesizer, offering the option of a full open-source framework. In the case of Yosys, only Verilog input files are allowed.

Evaluation. The second part of the program receives one of the fault configuration files describing the injection, together with an optional file with the input test vectors provided by the user. The tool employs an event-driven technique to simulate the circuit and determines the values of all primary outputs (including the *status* signals, *e.g.*, a signal indicating the detection of a fault). During the experiment, the same fault simulations are performed for every input. The tool reports the coverage for every test vector and a final overall result with the total number of faults, and the average coverage for the whole set of inputs. The tool reports all the non-detected faults per input test vector with the corresponding faulty output, as well as the ineffective faults.

VerFI's functionalities. The tool is designed to have high flexibility, allowing the users to define their fault model and evaluate their designs accordingly. The principal purpose of VerFI is to evaluate fault-injection countermeasures, but its functionality is not limited to this task. The user can mount simulated fault attacks over the actual netlist using the outputs generated by the fault simulations without the need to deploy the design on FPGA or ASIC. Nonetheless, the tool does not automate any attack; it only provides the means to perform them. Attacks like DFA, AFA [121], SEA [187] or SIFA, can make use of the results of VerFI to facilitate the key-recovery process. The user has full freedom to emulate different kinds of attacks and fault models.

6.3.3 Faults configuration

As mentioned in § 6.1.2, there are two injection mechanisms possible with VerFI: hierarchical fault injection and component-wise fault injection. Here, we discuss how with these methods any possible fault model presented in § 6.3.1 can be achieved.

Gate-input vs. gate-output fault models. By enabling faults in the primary input buffers, we can optionally activate the gate-input fault model. Since most of the countermeasures need to keep a strict hierarchy to avoid undesirable circuit optimizations when synthesized, it suffices to consider faults at inputs of sub-modules to deal with the gate-input fault model. The current version of the tool offers the possibility of injecting gate-input faults only at inputs of hierarchical sub-modules, not at the inputs of every single cell. This limitation is advantageous due to the lower number of faults that have to be considered.

Uniform vs. Biased fault models. The possibility to decide in the configuration files which elements to fault naturally allows the possibility to achieve a biased fault injection. Moreover, this also allows the evaluation of designs where just certain parts of the circuit are protected. It suffices to enable fault injection in all elements to select a uniform fault model. Additionally, by using the component-wise fault injection, we can perform an exhaustive fault analysis. Naturally, the amount of fault simulations overgrows with the number of components and with the number of faults to inject in the same fault simulation.

Univariate vs. Multivariate models and number of faults. Solely by deciding in which clock cycles VerFI can inject the faults, we can decide to operate in $\mathcal{M}_t$ or $\mathcal{M}_t^*$. Every module enabled for fault injection must be set to be faulted in the same clock cycle if $\mathcal{M}_t$ is desired. Then, the number of faults is decided per clock cycle, so in the hierarchical fault injection, the parameter N_t can be used to set this limit. In the component-wise injection, the user decides explicitly which component to fault and in which cycle, so up to t components should be enabled for fault injection in the same cycle.

Type. Finally, the user can easily set the fault type by specifying either 'SEU', 'SA0', or 'SA1'.

6.3.4 Evaluation

The primary goal of designing VerFI is to verify countermeasures against fault attacks on cryptographic implementations. The tool accepts two forms of detection mechanisms: if the circuit detects a fault, it either triggers an *abort* signal or sets the full output to zero to avoid revealing any information about the faulty output (methodology proposed in [5]). Hence, the faults are classified into three different groups, based on whether the cipheterxt output (ct) is faulty ($\mathcal{O}_f$) or not ($\mathcal{O}$), and on the abort signal $\mathcal{A}$:

1. *Detected ($\mathcal{F}_D$):* the design detects the injected fault using any of the aforementioned detection mechanisms, *i.e.*, $ct =?$ & $\mathcal{A} = 1$ or $ct = 0$.
2. *Non-Detected ($\mathcal{F}_N$):* the countermeasure does not detect the injected fault, and a faulty output is provided, *i.e.*, $ct = \mathcal{O}_f$ & $\mathcal{A} = 0$.
3. *Ineffective ($\mathcal{F}_I$):* the injected fault is not propagated to the output. In this case, the design does not detect the fault, but the output does not differ from the correct one, *i.e.*, $ct = \mathcal{O}$ & $\mathcal{A} = 0$.

The tool provides separated results for the coverage of effective faults and ineffective faults since the user could choose a countermeasure that protects against the first kind of faults but not the second. Thus, the detection coverage, given in Eqn. (6.1) does not take into account the ineffective faults. The subsequent section describes a verification procedure in case of SIFA attacks.

$$\mathcal{C} = \frac{\mathcal{F}_D}{\mathcal{F}_D + \mathcal{F}_N} \tag{6.1}$$

6.3.5 Statistical Ineffective Fault Attack

SIFA is fundamentally based on the fact that both effective (detected and non-detected) and ineffective faults appear. Hence, an algorithmic countermeasure to thwart SIFA could be an error *correction* mechanism, which prevents effective faults from appearing given a particular attacker model. Thereby error correction can prevent the root cause of SIFA.

In order to check for full protection against SIFA, we can compute a coverage that depends on the effective and ineffective faults (see in SIFA paper, page 16). To provide full protection against SIFA, 100% of the injected faults would have to be ineffective. The fact of having 100% ineffective faults would break the requirement: “require that there is some dependency between the observation

of an ineffective fault induction and the faulted intermediate value x" [76]. As a metric to address how possible is a SIFA attack, we use the *Fault Ineffectivity Rate (π)*, defined as follows [76]:

$$\pi = \frac{\mathcal{F}_I}{\mathcal{F}_D + \mathcal{F}_N}. \tag{6.2}$$

The ineffectivity rate refers to the probability of an ineffective fault to occur. An ideal SIFA would have a high ineffectivity rate. Nevertheless, if $\pi \to \infty$ ($\mathcal{F}_D + \mathcal{F}_N = 0$) we can say that a SIFA attack is not possible.[2] Additionally, if $\pi = 0$ ($\mathcal{F}_I = 0$), it means SIFA is also prevented. In this case, the countermeasure is designed to ensure that every fault is effective [70].

Countermeasures at the protocol level are also a viable alternative. Although VerFI has been primarily designed to analyze algorithmic countermeasures, a protocol based on limiting the number of queries (N) can be evaluated with our tool by computing the likelihood of a successful SIFA attack given π and N.

6.3.6 Infective Countermeasures

During the simulation of any particular fault, VerFI treats the design as a black box, only processing inputs and outputs of the top entity to control the simulation and check the fault detection, respectively. Despite parsing and emulating the actual netlist, the tool does not know what the functionality of every component or wire is - it has no semantic information.

Regular infective countermeasures do not provide any *status* signal for fault detection; they merely propagate the error *infecting* the state of the computation. Hence, the attacker only sees garbage output. The problem of most infective countermeasures is that a rightful user is unaware if the output is correct or random junk. Our tool suffers from the same problem since it is difficult to decide if the output is an exploitable faulty output or merely random.

This issue can be resolved with little intervention by the user, who may annotate a signal in the netlist by just adding the comment //Check_bits:<signalname> to assert whether the infection was triggered or not. Thus, if the output is incorrect and the assertion flag was triggered, the tool knows that the fault was detected, and the output is just a random string.

[2] Note that the designer should be cautious about inserting any kind of additional abort signal or detection mechanism that may give the attacker distinguishing information between effective and ineffective faults.

6.4 Case Studies

In order to test the functionality of the tool, we evaluated multiple protected implementations secured against either side-channel analysis, fault attacks, or even combined attacks from the literature. For side-channel verification, we tested protected implementations of Keccak using DOM [100], and several different masked Keccak S-boxes [31, 38, 69]. Regarding fault evaluation, we analyzed implementations of CAPA [158], M&M [74], the original ParTI LED implementation [168], and different Impeccable Circuits [5] implementations of Midori [15]. We include a description of the evaluation results below.

6.4.1 DOM Keccak

With our tool, we found a non-completeness flaw on the parallel implementations of DOM Keccak from [100], which we explained in § 3.5.1. During the design process, we analyzed every submodule with VerMI, successfully complying with both non-completeness and uniformity. It is then when composing the five Keccak permutations to get a complete round that the non-completeness failure appears.

We applied the same evaluation procedure with the corrected implementations presented in Fig. 3.20 and Fig. 3.21, which successfully fulfilled non-completeness. Finally, once the designs comply with the tool checks, we bring the final designs to the lab for the final verification step, which we presented in Fig. 3.22.

6.4.2 Masked Keccak S-boxes

We evaluated the uniformity of different first-order Keccak S-boxes from the literature [31, 38, 69], getting the same result as claimed in the original works. Fig. 6.6 shows the uniformity results for three different first-order Keccak S-box implementations from the previous three manuscripts.

The conclusion from the analysis matched the theory from the aforementioned papers. The first one is not uniform, as the frequencies are scattered among several different values. The second one achieves uniformity without any additional randomness with the frequency equal to 1 since the non-shared function is a permutation, and so is the shared version. The last one, with the same number of shares as the first one, uses four bits of randomness uniformly distributing the output with a frequency equal to 16.

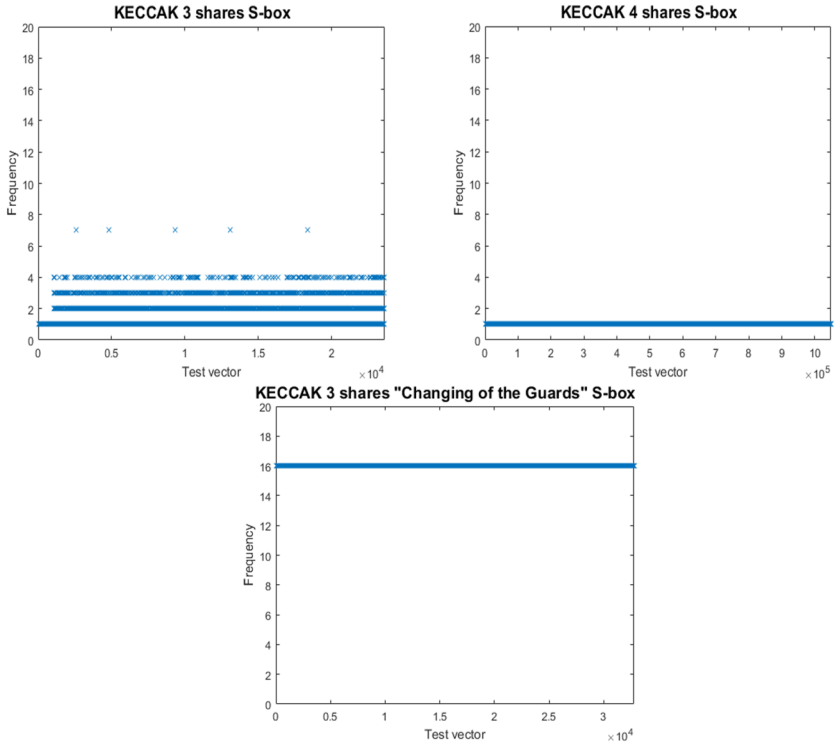


Figure 6.6: Uniformity analysis on KECCAK S-boxes [31, 38, 69] respectively from top to bottom

6.4.3 CAPA and M&M

By using VerFI, we conducted a thorough analysis of a hardware implementation of an 8-bit multiplier protected by CAPA with three shares, word-size $k = 8$, and number of mac tags $m = 1$. We corroborated the theoretical coverage of detected faults ($1 - 2^{-1 \cdot 8} = 0.99609$) with an experimental coverage of 0.99625. We were able to verify the coverage in multiple scenarios, including different fault models and a different number of faults injected, demonstrating the flexibility and accuracy of VerFI.

Additionally, we evaluated a design implementing infective countermeasures. As most of these countermeasures in the literature are broken, we chose M&M. We performed the experiments on the full AES M&M V2 implementation ($k = 8$, $m = 1$), and emulated a similar evaluation as performed in [74]. This experiment

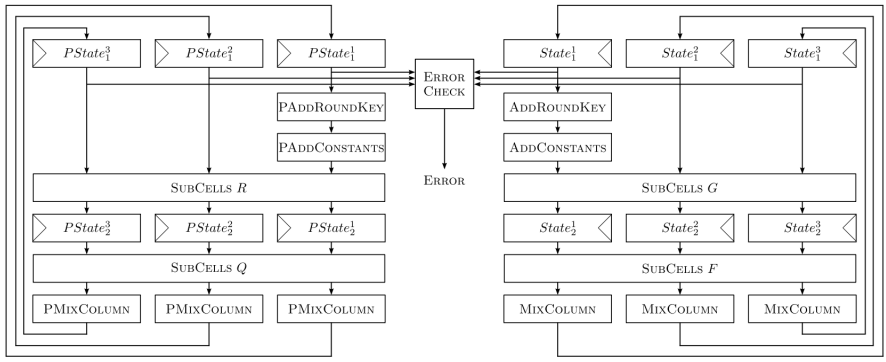


Figure 6.7: Structure of LED protected with ParTI, taken from [168].

demonstrates the applicability of VerFI to analyze infective countermeasures.

6.4.4 ParTI

LED [102] is an AES-like block cipher. To achieve fault tolerance ParTI [168] applies an Error-Detection Code to each nibble of the state. More precisely, each nibble is encoded with eight bits allowing two completely separated computations to take place, which we call the data path, illustrated on the right, and the parity path, illustrated on the left in Fig. 6.7. The error check can detect any error that is local to only one path of the computation and any error of Hamming weight of up to 3 overall 8 bits of each codeword. This property comes from the underlying extended Hamming code applied in ParTI [168].

Both the parity path on the left and the data path on the right are split into a three-share TI to ensure first-order SCA resistance. This requires the cubic S-Box to be split into quadratic functions G and F. Similarly, the parity path contains corresponding quadratic functions R and Q.

We had access to the original implementation of ParTI LED benchmarked in [168]. An initial evaluation with VerFI allowed us to spot an implementation mistake. The majority of the faults injected at the last two clock cycles of the computation were not detected since the check resolution happened one or two cycles after the output was released. The checking mechanism presented in [168] needs a careful unsharing that takes two cycles to complete, as shown in Fig. 6.8. This mechanism does not entail a problem during the computation. However, if the output is issued (triggering a *done* signal) right after the computation of

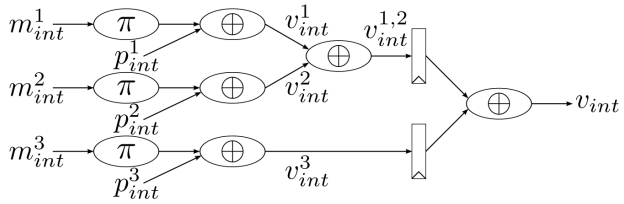


Figure 6.8: Computation and unmasking of the error check vector in ParTI for first-order security, taken from [168].

the last round (as it was the case), then the check for this round does not have enough time to finish. The authors have been notified and have provided an updated implementation.

We conducted further analysis on the corrected implementation, finding several non-detected faults within the fault model mentioned above. We managed to bypass the checking mechanism with 2-bit faults, i.e., of Hamming weight 2. They are injected either at the same clock cycle (univariate $\mathcal{M}_{t=2}$) or in subsequent clock cycles (multivariate $\mathcal{M}^*_{t=1}$). The non-linearity of the S-Box further propagates the faults, affecting more bits than the initially faulted, and producing a valid codeword that does not trigger the error detection. These findings confirm the suspicions raised in [5] since the given implementation of ParTI does not fulfill any of the properties proposed in that same work. With such non-detected faults, we can launch a DFA attack on the protected implementation. The details of the attack are given below.

DFA Attack on ParTI

We formulate criteria for faults to remain undetected by ParTI and subsequently describe two different DFA attacks leading to a full key recovery by exploiting undetected faults.

Undetected Faults. Due to the non-linearity of the S-Box stages, a fault limited to a few bits at their input may be propagated into a fault affecting more bits at the time of the check. Note that we stay within the gate-output fault model. Based on Fig. 6.7, faults injected into the AddRoundKey, can represent faults at the input of G. Similarly, the faults injected either at the G module or the registers placed between F and G would translate in faults at the input of F. The masking countermeasure within ParTI separates both data and parity paths into three shares. This countermeasure neither helps nor

Table 6.1: All undetected Two Bit Faults in each Nibble of ParTI-LED: Δ_J denotes the input fault of operation J.

Δ_G	Δ_F	Δ_R	Δ_Q	# non-detected	# ineffective
2	0	0	2	2	14
4	0	0	2	2	14
0	8	8	0	8	8
1	0	0	4	8	8
8	0	0	4	8	8
1	0	0	8	8	8

hinders the DFA as the fault can just be injected into an arbitrary share. Since the AddRoundKey blocks are only connected to the first share of G (resp. R), the adversary is limited to inject faults on this share in order to achieve a fault at the input of G. In Table 6.1, we summarize all faults (in hexadecimal) of Hamming weight two that lead to undetected faults at the time of the check. All presented fault locations have been found with our tool VerFI.

Once a faulty nibble passes the error check, there is no further possibility of detection in a later round as the EDC operates on each nibble individually. Hence, the described faults can be injected in an arbitrary round without detection.

Attacking Individual Nibbles. As LED is structurally very similar to AES, we can adapt the DFA demonstrated by Giraud [95] for key recovery. Since LED performs the MixColumns (MC) operation in the final round as well, we assume that the attacker computes equivalent keys k' corresponding to the actual cipher key via the relation

$$MC^{-1}(ct \oplus k) = MC^{-1}(ct) \oplus k', \text{ with } k = MC(k'),$$

where ct stands for ciphertext. For the sake of brevity, from here on, we refer to the elements before the last MC operation as ciphertexts while retaining the prime for clarity

$$ct'_c = MC^{-1}(ct_c) \text{ and } ct'_f = MC^{-1}(ct_f),$$

with ct_c and ct_f the correct and faulty ciphertexts, respectively.

We describe the attack corresponding to the third row of Table 6.1. For each nibble i, the attacker injects a non-detected fault (Δ_F, Δ_R) into the input of

F and R and determines the output. If the ciphertext is faulty, the attacker guesses one nibble of the key and computes back to the time of injection, thereby obtaining a distinguisher along the data path:

$$\Delta_F \stackrel{?}{=} F^{-1}(ct'_{f,i} \oplus kg'_i) \oplus F^{-1}(ct'_{c,i} \oplus kg'_i).$$

Similarly, the attacker can obtain a distinguisher along the parity path:

$$\Delta_R \stackrel{?}{=} R^{-1}(Q^{-1}(ct'_{f,i} \oplus kg'_i)) \oplus R^{-1}(Q^{-1}(ct'_{c,i} \oplus kg'_i)).$$

However, both distinguishers cannot be used jointly, as the MixColumn operation differs from the ParityMixColumn, thereby affecting the reference of all variables denoted with a prime.

The attacker must obtain at least two pairs to reduce the number of key guesses kg for one nibble to a unique one. Note that obtaining all 16 nibbles only derives the last half of all LED key bits (for 128-bit key). The same attack can be performed one step earlier, leading to the recovery of the other half of the key. In summary, the attacker is required to obtain a minimum of 64 correct and faulty ciphertext pairs in total to achieve a full key recovery. For simplicity, we restrict our analysis to an attacker who does not use computational power to obtain remaining key parts.

Attacking Column-wise. In the following, we demonstrate how to reduce the number of pairs to perform the attack. Mounting the previous attack one round earlier within the final step, it affects an entire column of the ciphertext. Hence, the attacker can gain information about four key nibbles at once. For the sake of simplicity, we ignore the ShiftRows operation and refer to the nibbles in one column with the indices 0,...,3. The adversary computes back an entire round and some part of the non-linear operation:

$$yd_{0,1,2,3}(ct') = MC_4^{-1}(S_4^{-1}(ct'_{0,1,2,3}))$$

$$zd_{0,1,2,3}(ct') = F_4^{-1}(yd_{0,1,2,3}(ct')),$$

where MC_4 corresponds to mixing only one column, and F_4 to the parallel application of the non-linear function to four nibbles.

Without loss of generality, we assume an injection of fault Δ_F into the first nibble, then the distinguisher is

$$(\Delta_F, 0, 0, 0) \stackrel{?}{=} zd_{0,1,2,3}(ct'_c) \oplus zd_{0,1,2,3}(ct'_f).$$

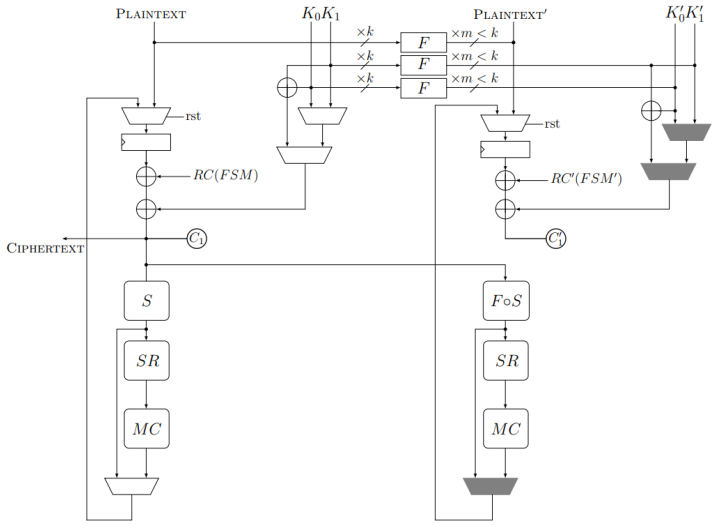


Figure 6.9: Implementation of round-based Midori for a small degree of redundancy, taken from [5].

Further, it is even possible to find a distinguisher without knowledge of F_4 and exact knowledge of the injected fault by considering the equations

$$0 \overset{?}{\neq} yd_0(ct_c') \oplus yd_0(ct_f'), \quad 0 \overset{?}{=} yd_i(ct_c') \oplus yd_i(ct_f'),$$

for $i = 1, 2, 3$. This needs, on average, two pairs of (ct_c, ct_f) to determine a unique key guess for four nibbles. Hence, the adversary can extract each half of the key with only eight such pairs (resp. 16 pairs in total to extract the full key). We verified the feasibility of this attack by using VerFI to generate the 16 necessary pairs and determining the correct key accordingly.

6.4.5 Impeccable Circuits

A methodology that enables the secure and practical implementation of code-based CED schemes in the presence of fault propagation is suggested in [5].

In order to evaluate the soundness of the fault models considered in [5], we targeted Impeccable Circuits implementations of Midori [15] and a toy cipher. The authors have considered the gate-output fault model $\mathcal{M}_{\mathcal{O}}$ in their analyses, and designed the schemes and proved the theorems accordingly. Utilizing

VerFI, we subjected the implementation to a stronger adversary model, *i.e.*, gate-input fault model $\mathcal{M}_{\mathcal{I}}$. Using this model, we were able to bypass the checking mechanism of several implementations. Fig. 6.9 presents the scheme of a round-based Midori protected with a small degree of redundancy. The implementation is carefully designed by using the independence property and taking care of the fault propagation [5], as well as placing the checks accordingly, to avoid further propagation of the faults. In this particular case, the only block susceptible to propagate a fault is the S-Box. No fault injected in any of the gates inside the S-Box would propagate further to create a valid codeword. The check is placed right before the S-Box, so no previously injected fault would reach its inputs. This investigation confirmed the consistency of the claim of Impeccable Circuits based on its underlying $\mathcal{M}_{\mathcal{O}}$ model. Instead, by injecting a fault at the input of the S-Box, *i.e.*, $\mathcal{M}_{\mathcal{I}}$ model, the faults are further propagated to get a valid codeword, bypassing the checks, similarly to what happened in the case of the ParTI LED. This experiment practically illustrates how, by injecting faults at the gate inputs, we achieved a more powerful fault model.

6.5 Results

In this section, we present the analysis results reported by both tools VerMI and VerFI for several state-of-the-art hardware implementations protected against physical attacks. Additionally, we present both tools' performance. Experiments were run in an Intel Core i5-4590 CPU, 3.3GHz clk, and 7.6 GB RAM running a 64-bit Linux CentOS7.

6.5.1 Evaluation

VerMI

Table 6.2 gathers the results for several implementations analyzed: protected AND gates from the literature, various sharing schemes of the Keccak S-box and AES S-box, and few whole round-based Keccak implementations.

Contrary to [43], VerMI is limited to univariate analysis (probes in the same cycle). However, this results in a significantly improved performance, which allows testing complete ciphers, while [43] is limited by the inner complexity. The possibility to check non-completeness on a whole cipher led us to find a flaw in the round-based Keccak implementation from [100], where only analyzing

Table 6.2: Verification Tool benchmark

Design	Cells		Variables			1st N-C		2nd N-C		3rd N-C		Uniformity	
	Comb.	Regs.	Ins	Rand	Outs	Time	Result	Time	Result	Time	Result	Time	Result
Shared AND gates													
Trich [177]	7	0	4	1	2	2.2 ms	✗	2.5 ms	✗	2.5 ms	✗	442 μs	✓
ISW [108]	7	0	4	1	2	2.2 ms	✗	2.6 ms	✗	2.4 ms	✗	365 μs	✓
TI [147]	12	0	6	0	3	2 ms	✓	2.7 ms	✗	2.6 ms	✗	739 μs	✗
DOM [98]	8	4	4	1	2	1.7 ms	✓	2.2 ms	✗	2.5 ms	✗	189 μs	✓
Shared Keccak S-boxes													
DOM [100]	45	20	10	5	10	1.3 ms	✓	2.6 ms	✗	10.1 ms	✗	1.6 s	✓
TI3sh. [31]	99	0	15	0	15	1.9 ms	✓	2.8 ms	✗	11 ms	✗	1.7 s	✗
CG3sh. [69]	104	0	15	4	15	2 ms	✓	4.5 ms	✗	9.8 ms	✗	29 s	✓
TI4sh. [38]	82	0	20	0	20	1.2 ms	✓	4.4 ms	✗	16.3 ms	✗	62 s	✓
Shared $d+1$ AES S-boxes													
2sh. [65]	862	144	16	64	16	4.8 ms	✓	55.7 ms	✗	417 ms	✗	-	-
3sh. [65]	1.95k	270	24	162	24	6.5 ms	✓	106 ms	✓	4.03 s	✗	-	-
Shared round-based Keccak Implementations													
DOM [100]	2.7k	800	400	200	800	22.86 ms	✗	7.28 s	✗	45.2 m	✗	-	-
TI3sh.	4.2k	0	600	0	600	37.82 ms	✓	11.12 s	✗	58.8 m	✗	-	-
TI4sh.	6.3k	0	800	0	800	60.98 ms	✓	21.47 s	✗	150 m	✗	-	-
TI6sh.	17k	0	1200	0	1200	137.28 ms	✓	87 s	✓	14.8 h	✗	-	-
Related work [43]												Analysis time w. glitches/Result	
DOM AND	8	2	-	-	-	-	-	-	-	-	-	$\leq$ 2 s	✓
DOM Kec. SB	50	10	-	-	-	-	-	-	-	-	-	$\leq$ 20 s	✓
DOM AES SB	536	208	-	-	-	-	-	-	-	-	-	$\leq$ 5-10 h	✓

the S-box, the problem would not be found since the surrounding operations cause it.

VerFI

We report the practical results obtained using VerFI for implementations of ciphers protected by ParTI, CAPA, Impeccable Circuits, and M&M (cf. Table 6.3). We had access to both original implementations of ParTI and Impeccable Circuits from [168], and [5], respectively.

We analyzed all designs within the fault model proposed in their original work, except for the examples in rows 11 and 12, which we investigated in a stronger fault model ($\mathcal{M}_{\mathcal{I}}$ instead of $\mathcal{M}_{\mathcal{O}}$).

We simulated the CAPA designs within the gate-input fault model $\mathcal{M}_{\mathcal{I}}$, and without a restriction on the places to inject the fault ($\mathcal{M}_{\mathcal{U}}$). We inject 1, 8, and 24 faults in separate experiments (rows 1, 2, and 3, respectively, of Table 6.3). In the first case, the observed coverage is very close to the theoretical claim. When a higher number of faults are injected, the coverage increases due to a higher probability of the detection mechanism being triggered. It is interesting

Table 6.3: Performance evaluation of different fault and combined countermeasures from the literature, including fault model, number of simulations, inputs and faults, injections cycles, and coverage.

Design	Fault model				# Simuls	# Inputs	# Faults	Cycle(s)	$\mathcal{C}_{theory}$	$\mathcal{C}_{tool}$
	$\mathcal{I}/\mathcal{O}$	$\mathcal{U}/\mathcal{B}$	$\mathcal{M}_t/\mathcal{M}_t^*$	Type				(Total)		
CAPA [158] 3 shares 8-bit multiplier (k=1)										
1 Idem	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{U}$	$\mathcal{M}_1$	SEU	100	500k	1	1,2,3(3)	0.99609	0.9973
2 Idem	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{U}$	$\mathcal{M}_8^*$	SEU	100	500k	8	1,2,3(3)	0.99609	0.9997
3 Idem	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{U}$	$\mathcal{M}_{24}^*$	SEU	100	500k	24	1,2,3(3)	0.99609	1
4 Idem	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_{3\to10}^*$	SEU	560 → 12870	20	3 → 10	1,2,3(3)	0.99609	0.99625
ParTI [168]										
5 Idem	$\mathcal{M}_\mathcal{O}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_2$	SEU	1k	4	2	93(98)	1	0.994
6 Idem	$\mathcal{M}_\mathcal{O}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_1^*$	SEU	10k	4	2	93.94(98)	1	0.9977
Impeccable Circuits [5] with code of length n, dimension k, and distance d as $[n,k,d]$										
7 Toy Cipher [5,4,2]	$\mathcal{M}_\mathcal{O}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_1$	SEU	120	10	1	14(16)	1	1
8 Toy Cipher [6,4,2]	$\mathcal{M}_\mathcal{O}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_1^*$	SEU	42k	10	2	>9(16)	1	1
9 Toy Cipher [7,4,3]	$\mathcal{M}_\mathcal{O}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_2^*$	SEU	100k	10	4	>9(16)	1	1
10 Toy Cipher [8,4,4]	$\mathcal{M}_\mathcal{O}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_3^*$	SEU	150k	10	6	>9(16)	1	1
11 Toy Cipher [5,4,2]	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_1$	SEU	20	3	1	13(16)	-	0.79
12 Toy Cipher [8,4,4]	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_2^*$	SEU	1.4M	3	2	>9(16)	-	0.999979
13 Midori Enc [8,4,4]	$\mathcal{M}_\mathcal{O}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_3$	SEU	200k	10	3	14(18)	1	1
14 Midori Enc [8,4,4]	$\mathcal{M}_\mathcal{O}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_3^*$	SEU	400k	10	6	>12(18)	1	1
AES M&M V2 [74] 2 shares (d=1), and single alpha (k=1)										
15 Idem	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_1$	SEU	1	10k	1	>199(254)	0.99609	0.9957
16 Idem	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_8^*$	SEU	1	10k	8	>199(254)	0.99609	1
17 Idem	$\mathcal{M}_\mathcal{I}$	$\mathcal{M}_\mathcal{B}$	$\mathcal{M}_{1,2,4,8,12}$	SEU	1	5k	1,2,4,8,12	225(254)	0.99609	0.9959

to see how the injection of a single fault in this model is more harmful than the injection of multiple faults. We perform an additional experiment (row 4), this time within the model $\mathcal{M}_\mathcal{I} \cap \mathcal{M}_\mathcal{B}$, where we targeted exclusively one variable (ϵ) of the multiplication and its respective tag (τ^ϵ). This analysis entails a total of 16 locations, which we exhaustively check for combinations of a different number of faults, ranging from 3 to 10. The number of simulations increases from 560 to 12870. The result is the average of the eight experiments, achieving a coverage extremely close to the theoretical one. Since we target value and tag of a single variable, the resulting coverage is the probability of forging a correct tag. We attribute the small differences between theoretical and experimental coverages to the impossibility to exhaust all inputs.

Designs protected with ParTI are analyzed within the model $\mathcal{M}_\mathcal{O} \cap \mathcal{M}_\mathcal{B}$, with up to two-bit faults injected in each case, excluding the control datapath and the error check. ParTI fails to provide the promised full coverage. Nevertheless, the coverage is still comparably high, ranging between 0.994 and 0.9977. In the first experiment (row 5), the attack focuses on a single share of the first stage of the SBox, for both datapaths. In the second experiment (row 6), we additionally allowed the second stage of the SBox to be faulty in the subsequent clock cycle.

We evaluated several implementations protected with Impeccable Circuits, including a full Midori encryption function and a toy cipher. The experiments are conducted under the model $\mathcal{M}_{\mathcal{O}} \cap \mathcal{M}_{\mathcal{B}}$, restricting the number of faults per clock cycle to t (rows 7 - 14). The tool reports full coverage for all of them, as claimed by the respective authors. Note that this time, the faults could be injected at data processing, the control logic, and error check modules. Additionally, we investigated two implementations further within the $\mathcal{M}_{\mathcal{I}}$ model (rows 11 and 12). Although a high level of coverage is achieved, the designs do not provide full coverage against such an adversary model. This result does not contradict their security guarantees since protection under this model is never claimed.

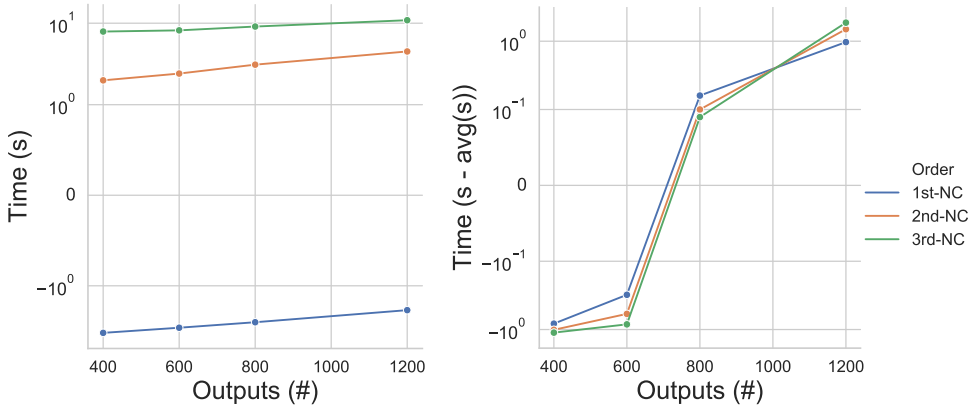
Finally, we present the experiments for M&M AES (rows 15, 16, 17 of Table 6.3). The first two experiments inject respectively one and eight faults, focused on the most sensitive modules, and the last two rounds of the computation. In the last experiment, we perform a component-wise injection, selecting similar components as in [74], and performing different injections of 1, 2, 4, 8, and 12 number of faults. The resulting coverage of this experiment is the average of the result of every injection. Since the security claims of this scheme include that α is never repeated for more than one computation, every experiment is performed with a single fault simulation for each input (# Simuls = 1) to comply with this requirement. We observe that all non-detected faults for the first experiment (row 15) occur when $\alpha = 0$, which happens with probability 2^{-8}. The coverage obtained with VerFI is exceptionally close to this probability. We obtain full coverage in the second experiment (row 16) since injecting multiple random faults increases the chance of one triggering the infection mechanism. The final experiment (row 17) also reports a coverage very close to the theoretical one. Even though this attacker is more potent since specific points of attack are chosen, it is still not better than guessing α.

6.5.2 Performance

The complete set of tools is fully implemented in C++ for high performance. Hereby, we show how non-completeness evaluation time evolves with the number of outputs to analyze, how simulation time is affected by the circuit toggles, and how the number of faults per injection affect simulation time.

VerMI

Non-completeness. Fig. 6.10 presents the time that the tool needs to evaluate first-, second-, and third-order non-completeness for the full round of several



(a) Regular plot

(b) plot centered to zero

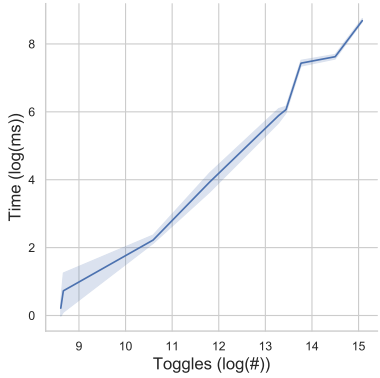
Figure 6.10: Non-completeness evaluation, time versus number of output sensitive bits.

Keccak implementations with different numbers of shares to see the effect of the different number of outputs on the execution time. The first picture shows the actual time for the different orders in the logarithmic scale. For the sake of clarity, a second picture is provided with the plots centered to zero. As it can be seen, not only the actual time of higher orders is more significant, but also the higher the order of the analysis, the faster the time grows with the number of output bits.

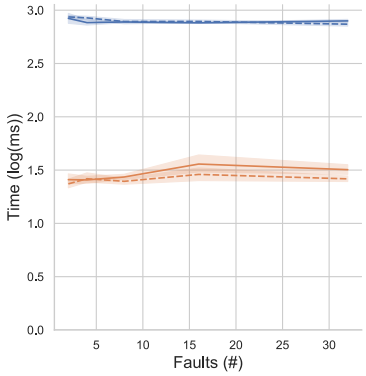
Uniformity. It is trivial to see that the evaluation time for uniformity is exponential with the number of inputs, including the random variables. For a fixed number of inputs, since and event-driven simulation is performed, the processing time is proportional to the number of toggles during the simulation. The largest design analyzed was a six shares 2nd-order Keccak S-box, which meant evaluating 2^{30} test vectors, and which took almost two days.

VerFI

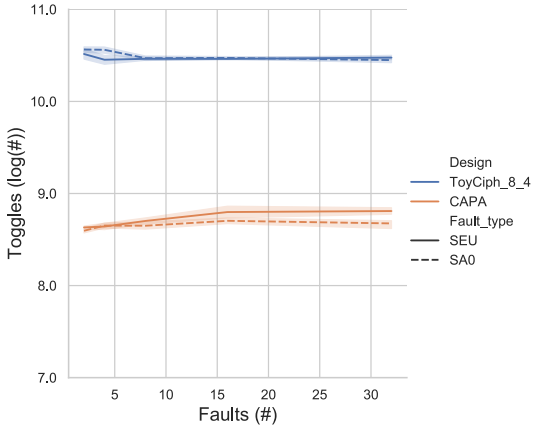
As mentioned before, simulation is based on the propagation of signal changes. Hence, the execution time is independent of the area of a design and proportional to the number of toggles during the simulation (Fig. 6.11a). In the case of fault evaluation, the circuit is entirely simulated in time for all the necessary clock



(a) Toggles vs. Time



(b) Faults vs. Time



(c) Faults vs. Toggles

Figure 6.11: Evolution of VerFI's execution time with respect to the number of toggles and the number of faults per injection.

cycles. Naturally, the more cycles to simulate, the longer the execution time. Noticeably, the number of total simulations proportionally increases execution time, while a different number of faults injected per simulation (N_f) does not significantly alter the execution time for SEU and SA0 faults (Fig. 6.11b). Similarly, the number of toggles remains indifferent to the total number of faults (Fig. 6.11c).

Finally, we address the number of simulations for exhaustive checking. Given

N_e as the total number of cells enabled to be faulty in a particular cycle and N_f as the number of faults per injection, the number of simulations to exhaustively check all combinations is given by the rapidly growing term $\binom{N_e}{N_f}$.

6.6 Conclusions

In this chapter, we have presented VerMFi, the combination of the works where VerMI and VerFI are proposed. It is a set of verification tools to help in automating the security evaluation of cryptographic hardware implementations. VerMI can address univariate passive security of any logic circuit, even in the presence of glitches, by checking non-completeness and uniformity.

We have introduced the concept of Cryptographic Fault Diagnosis, a structured revision of fault diagnosis concepts known from reliability testing according to their relevance to cryptographic implementations. Moreover, we have proposed a consolidated characterization of cryptographic fault models, showing the importance of the Gate-input fault model by subjecting multiple implementations of Impeccable Circuits [5] to this stronger fault model, and finding faults not detected.. VerFI facilitates the practical application of our theory, being the first evaluation tool for cryptographic fault analysis working directly on the gate-level netlist of hardware designs.

Finally, we have demonstrated the value and usability of VerMFi by finding several flaws in various implementations from the literature. VerMI spotted a weakness in the round-based KECCAK implementation from Gross *et al.* [100]. With FerFI, we identified implementation and algorithmic flaws in an instance of LED protected by ParTI [168] and performed a practical attack on a fixed implementation leading to a full-key recovery in the presence of adversaries covered by ParTI with the tool's information.

Chapter 7

Conclusion

In the last chapter of this thesis, we first give a summary of the previous chapters, and then we refresh the research questions posed in Chapter 1, together with the conclusions in the form of answers to these questions. To finalize, we provide directions for future work.

7.1 Summary and conclusions

We started this thesis with a brief outline of the multiple threats for the ever-increasing number of smart devices, which manifests the need for countermeasures against these menaces. Moreover, the verification of the correct implementation of these countermeasures is similarly essential. Once the problem was framed, we presented some preliminary concepts, in which the subsequent contributions are based. Then, we followed with the main chapters, each derived from a distinct research question. First, we proposed several unprotected implementations of different cryptographic algorithms, including the SHA-3 underlying permutation Keccak, the Authenticated Encryption algorithm Ketje, and the actual SHA-3 hash function. Together with this, we proposed several Side-Channel Analysis protected implementations, in addition to several concepts that are necessary or helpful to achieve correct implementation of SCA countermeasures. Secondly, we studied how these countermeasures perform against hybrid attacks, in particular, Fault Sensitivity Analysis. Subsequently, we built on the notions from previous chapters, adding the concepts of Fault Analysis into the studies, and presented two combined countermeasures against SCA and FA. Finally, we introduced the set of verification tools that we

developed in tandem with previous chapters. These tools implement checks for the critical notions noticed from a design point of view, and at the same time, helped to verify the secured implementations, providing feedback on the correct application of the countermeasures.

Below, we formulate the conclusions extracted from this thesis:

Question 1. *What are the implications and trade-offs of choosing a $td+1$ vs. a $d+1$ masking scheme? How can we secure an unrolled low-latency implementation against Side-Channel Analysis?*

We observed in Chapter 3 that applying $d+1$ masking schemes as CMS or DOM leads to using a smaller number of shares than TI to achieve the same security level, which generally translates into more area-efficient designs. However, the former schemes need extra fresh randomness and an extra layer of registers to keep the non-completeness property safe, which implies a supplementary clock cycle when securing a non-linear operation. Additionally, CMS and DOM like schemes must have independent inputs, which means one has to be careful when composing non-linear operations even with linear ones. In the case where linear operations precede non-linear ones, generally, an additional layer of registers and randomness are needed to ensure that dependencies among inputs are not created.

In our study, the first-order secure DOM implementation NC-Keccak (cf. Fig. 3.21) turned out to be more expensive in area compared to the one presented in § 3.4.1, which uses 3-share TI. This result is counterintuitive with a priori expectations mentioned in the previous paragraph. For the design of NC-Keccak, due to the input dependencies problem and the synchronization stage from the DOM scheme, we needed to add two new layers of registers that make the design more complex in terms of area than the counterpart using TI. In this specific situation, we do not need extra refreshing to mask the dependencies because the neighbor bits in the Keccak state are independent of each other and refresh themselves when mixed. Even though this instance grows in circuit complexity, the advantage of the pipelined implementation for the DOM sharing is that the maximum frequency increases. In the alternative case, provided a serial implementation, the conflicting bits found in the round-base design (cf. § 3.5.1) would be registered several times before arriving at the non-linear layer. Hence, the additional register to prevent dependencies coming from the linear layers would not be necessary. For higher security orders, the difference in shares between $td+1$ and $d+1$ schemes becomes bigger, the higher the order, which means that the contribution of the linear modules to the area is also greater. Hence, $d+1$ designs become then generally less complex in terms of area for higher-order designs.

As general advice, the supplementary layer of registers can be avoided by implementing first the non-linear operation, followed by the linear operations. The linear operations ensure no further combination within shares, so the designer can more confidently compose the linear layer with the non-linear ones this way. Note that this is not proven, it is meant as a guideline for helping on the task of implementing secure hardware designs.

Another example where the input dependencies demanded additional resources in order to preserve the security, is the combined scheme M&M, presented in Chapter 5. When implementing Optimization 1 for M&M V1 (cf. § 5.3.2), linear operations had to precede the calculation of x^5, so the addition of extra refreshing and registers layer was unavoidable.

Taking into account all these conclusions, a methodology for low-latency masking based on $td + 1$ masking was devised. This methodology is meant for securing unrolled implementations, but it is similarly suitable for securing complex functions with high algebraic degree (cubic or higher) targeting low-latency.

Question 2. *Can Side-Channel Analysis countermeasures prevent hybrid attacks?*

From the study performed in Chapter 4, we saw that indeed some SCA countermeasures provide resistance against Fault Sensitivity Analysis under the given assumptions. In particular, it is the non-completeness property that ensures this protection. This conclusion does not guarantee that every hybrid attack is thwarted with this kind of countermeasure since there is not a fixed nature for hybrid attacks. It is important to note that FSA is not a combined attack, and surely, SCA countermeasures cannot prevent fault attacks. The difference between hybrid attacks and combined attacks is not black and white. Combined attacks combine fault injection and probing, whereas hybrid attacks only use one of them, fault injection *or* probing. Then, they use Side-Channel Analysis analysis techniques when a fault is injected to retrieve the secret, or vice-versa. In § 5.4.3, we presented several fault and combined attacks. All the fault attacks have in common that they check the resulting ciphertext, to see whether the result is faulty or not. They either learn something directly from this observation or apply cryptanalysis techniques to retrieve the secret key. In FSA, only faults are injected, and even though the attack needs to check whether the ciphertext was faulty, the secret is revealed by exploiting the side-channel produced by the fault employing SCA techniques. Thus, this is the underlying reason why an SCA countermeasure can prevent this attack.

Question 3. *How can we design provably secure and practically feasible protection mechanisms against combined SCA and FA, and efficiently implement them?*

Algorithmic SCA countermeasures originally derive from MPC protocols, achieving provable security. Inspired by this result, in Chapter 5, we looked at actively secure MPC protocols to achieve provable security against both passive and active adversaries combined. The investigations resulted in the Tile-Probe-and-Fault model, a highly robust model that encapsulates a potent attacker. The combined provable security and the sturdy adversarial model resulted in a highly expensive implementation, both in terms of area and randomness. A considerable amount of resources were pushed to the preprocessing or offline phase of the computation to comply with the security requirements. With CAPA, we aimed to establish an exceptionally robust countermeasure, meant as a starting point for subsequent works. The model can be adapted to the needs of the targeted application, resulting in a more robust scheme.

With this idea in mind, we designed the next scheme by combining two independent countermeasures, where a particular masking methodology is extended with information-theoretic MAC tags to provide additional fault protection. This merge of two independent countermeasures entails a reduction of the security claims, given that combined security cannot be proven anymore. Additionally, the adversarial model is also relaxed, where the new adversary is more constrained on the probing capabilities. The outcome of this investigation was M&M, with which we achieved a more efficient implementation both in terms of area and randomness.

From this research, we can see how important it is for a security designer to consider the different trade-offs between the provided security guarantees, the adversarial model, and the resources needed. By adapting the adversarial model to a more particular case or application, it is possible to obtain a more efficient implementation.

Question 4. *How can physical countermeasures be efficiently and practically verified in an automated fashion? Which are the metrics that we have to check?*

With the examples presented in § 6.4, we saw the importance of a comprehensive verification and the need for tools to facilitate this task. We learnt that there is a considerable amount of tools in the literature at different levels of abstraction, which help with the verification at different stages of the design process. We conclude that all of them are equally important, highlighting the importance of verifying the security of a design at every stage of the design flow. These tools present different trade-offs regarding exhaustiveness and accuracy. More formal

tools, taking advantage of abstractions of the reality in different models, can achieve more exhaustive verifications at the expense of less practicality, due to failing to take into account different physical properties in the model. On the other hand, with more practical evaluation methods, it is not possible to cover every case, and hence the security assessment is given with a certain degree of confidence. Nevertheless, the result is more conclusive.

In our particular case, we designed two tools intended to be highly practical. The first one, VerMI, evaluates two specific properties of TI. The second one, VerFI, performs fault simulation under a wide variety of fault models, and with extremely flexible injection mechanisms. In both cases, it is respectively unfeasible or difficult to get an exhaustive security evaluation. However, the characteristic practicality allowed us to analyze full implementations, which led us to find several flaws in two different architectures published in the literature.

7.2 Future work

In this final section, we propose several directions for future research within three different categories: SCA countermeasures, FA and combined countermeasures, and verification procedures.

Low-latency and low-power SCA countermeasures

Most of the literature regarding the design of SCA protected implementations has continuously focused on optimizing the circuit complexity. Provided the unceasing growth of the number of transistors within a chip, the area concerns diminish. Thus, we believe that Side-Channel Analysis countermeasures for low latency will be more demanded in the coming years. Besides the work presented in this thesis in § 3.5.2, only one other paper covers the topic of masking for low latency implementations [96]. Compared to the conventional masking literature, this is a profoundly new topic. One could start answering questions like: “Can we create more efficient tailored schemes by using new masking methodologies?”, or “How do we need to refresh low latency masking schemes?”. Additionally, it would be also interesting to study low-energy masked implementations and if there are algorithmic optimizations that we can employ to improve the energy trade-off.

Stronger and more efficient FA and combined countermeasures

Regarding Fault Analysis, the new SIFA [77] attack was recently published, which means efficient countermeasures need to be proposed. Few works are already appearing on this matter, but unquestionably further investigations are needed.

Up until very recently, countermeasures against SCA and FA have been studied mostly independently. However, attackers are assumed to have direct access to cryptographic devices under consideration. One may think that they can launch a fault attack if the device is protected against SCA and vice-versa. It certainly depends on the application, but we believe that the research of combined countermeasures is an essential topic towards the future. The first step could be to look into the preprocessing stage of CAPA, which is the most expensive part of the design, and investigate possible optimizations.

Further inquiries could be the design of a similar scheme without any offline phase and the security implications. Finally, another exciting subject to look at is the possibility of finding strategies to make M&M, or other schemes where two independent countermeasures are combined, provably secure against combined attacks. "Which conditions make a scheme provable secure against combined attacks?"

Optimizing VerMFi, and defining combined verification procedures

As discussed through the thesis, it is imperative to integrate different verification procedures into the design flow of implementations secured against physical attacks. For that, efficient verification tools are needed.

Firstly, as future work regarding our set of tools, the immediate next step for VerMI would be to implement support for multivariate analysis and a more thorough security evaluation at higher-order. Additionally, looking into strategies to optimize the uniformity check is also an interesting direction for future work. Regarding VerFI, the next step would be to implement faults in wires to achieve full granularity and full power with the faults injected, thus achieving gate-input faults in every cell. Nevertheless, this will lead to a significant increase in analysis complexity, as many more faults should be simulated. Additionally, it is possible to significantly enhance the performance of the current version by implementing parallel fault simulation. The idea is to transform the currently-implemented bit-wise operations to operate on 256-bit words by utilizing AVX2, where each bit of the word would correspond to a different fault simulation, allowing us to perform 256 fault simulations

simultaneously. We could also investigate sensitized paths to considerably reduce the number of faults to test.

Secondly, for verification procedures in general, there is still a need for more efficient formal verification with a realistic model, also taking into account physical effects like coupling [62]. As an ultimate target, we need formal and practical tools capable of handling full implementations and giving exhaustive security guaranties.

Finally, as a next step, we need verification procedures to analyze the combined security of a design as a whole, instead of analyzing the protection to SCA and FA separately. A starting point can be to check how a TVLA test reacts when a fault is injected in the computation. Undoubtedly, the complexity of this verification process would be extremely high, given that a separate TVLA would have to be performed for all the different possible faults.

Appendix A

SCA Keccak miscellaneous

A.1 Keccak Round Steps

A.2 Second-order Masking

A.2.1 Threshold Implementation (TI)

$$
\begin{aligned}
z_1 &= a_1 \oplus b_1c_1 \oplus b_1c_5 \oplus b_5c_1 & z_2 &= a_2 \oplus b_2c_2 \oplus b_1c_2 \oplus b_2c_1 \\
z_3 &= a_3 \oplus b_3c_3 \oplus b_1c_3 \oplus b_3c_1 & z_4 &= a_4 \oplus b_4c_4 \oplus b_1c_4 \oplus b_4c_1 \\
z_5 &= a_5 \oplus b_5c_5 \oplus b_2c_5 \oplus b_5c_2 & z_6 &= b_2c_3 \oplus b_3c_2 \\
z_7 &= b_2c_4 \oplus b_4c_2 & z_8 &= b_3c_4 \oplus b_4c_3 \\
z_9 &= b_3c_5 \oplus b_5c_3 & z_{10} &= b_4c_5 \oplus b_5c_4
\end{aligned}
\tag{A.1}
$$

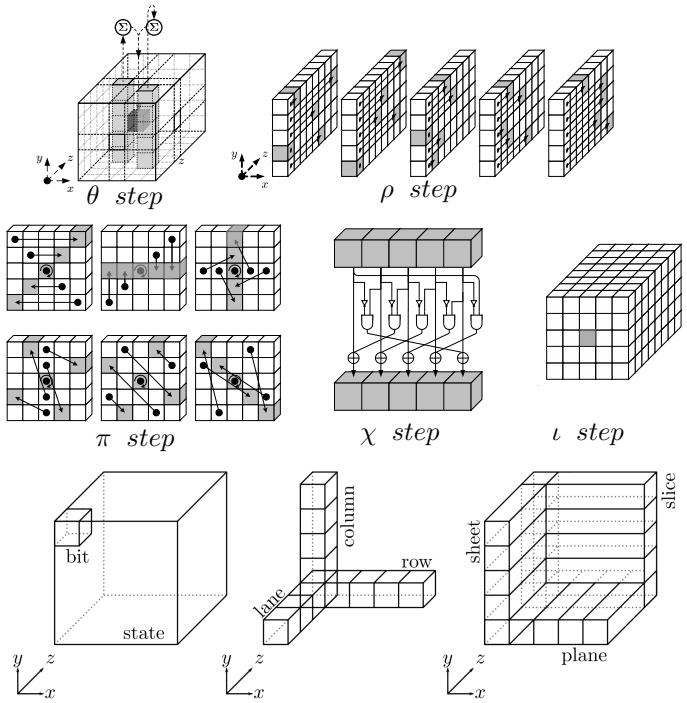


Figure A.1: KECCAK-f[200] state and steps [32].

A.2.2 Consolidated Masking Scheme (CMS)

$$
\begin{aligned}
z_1 &= a_1 \oplus b_1 c_1 & z_2 &= a_2 \oplus b_1 c_2 \\
z_3 &= a_3 \oplus b_1 c_3 & z_4 &= b_2 c_1 \\
z_5 &= b_2 c_2 & z_6 &= b_2 c_3 \\
z_7 &= b_3 c_1 & z_8 &= b_3 c_2 \\
z_9 &= b_3 c_3
\end{aligned}
\tag{A.2}
$$

A.3 Second-Order Non-completeness Failure in Round-Based DOM Implementation

First bit

$$\chi^{-1}: \quad S_2^{\chi}[0,0,7] \quad \to S_2^{\pi}[0,0,7] \oplus S_2^{\pi}[1,0,7]\, S_2^{\pi}[2,0,7]$$

$$\begin{aligned}
\pi^{-1}: \quad & S_2^{\pi}[0,0,7] && \to S_2^{\rho}[0,0,7] \\
& S_2^{\pi}[1,0,7] && \to S_2^{\rho}[1,1,7] \\
& S_2^{\pi}[2,0,7] && \to S_2^{\rho}[2,2,7]
\end{aligned}$$

$$\begin{aligned}
\rho^{-1}: \quad & S_2^{\rho}[0,0,7] && \to S_2^{\theta}[0,0,7] \\
& S_2^{\rho}[1,1,7] && \to S_2^{\theta}[1,1,3] \\
& S_2^{\rho}[2,2,7] && \to S_2^{\theta}[2,2,4]
\end{aligned}$$

$$\begin{aligned}
\theta_1^{-1}: \quad & S_2^{\theta}[0,0,7] && \to S_2[0,0,7] \oplus \bigoplus_{y'=0}^{4} S_2\left[4,y',7\right] \oplus \bigoplus_{y'=0}^{4} S_2\left[1,y',6\right] \\
& S_2^{\theta}[1,1,3] && \to S_2[1,1,3] \oplus \bigoplus_{y'=0}^{4} S_2\left[0,y',3\right] \oplus \sum_{y'=0}^{4} S_2\left[2,y',2\right] \\
& S_2^{\theta}[2,2,4] && \to S_2[2,2,4] \oplus \bigoplus_{y'=0}^{4} S_2\left[1,y',4\right] \oplus \sum_{y'=0}^{4} S_2\left[3,y',3\right]
\end{aligned}$$

Second bit

$$\chi^{-1}: \quad S_6^{\chi}\,[2,0,5] \quad \to S_1^{\pi}\,[4,0,5]\,S_3^{\pi}\,[3,0,5]$$

$$\pi^{-1}: \quad S_1^{\pi}\,[4,0,5] \quad \to S_1^{\rho}\,[4,4,5]$$

$$\qquad\quad S_3^{\pi}\,[3,0,5] \quad \to S_3^{\rho}\,[3,3,5]$$

$$\rho^{-1}: \quad S_1^{\rho}\,[4,4,5] \quad \to S_1^{\theta}\,[4,4,7]$$

$$\qquad\quad S_3^{\rho}\,[3,3,5] \quad \to S_3^{\theta}\,[3,3,0]$$

$$\theta_1^{-1}: \quad S_1^{\theta}\,[4,4,7] \quad \to S_1\,[4,4,7] \oplus \bigoplus_{y'=0}^{4} S_1\left[3,y',7\right] \oplus \bigoplus_{y'=0}^{4} S_1\left[0,y',6\right]$$

$$\qquad\quad S_3^{\theta}\,[3,3,0] \quad \to S_3\,[3,3,0] \oplus \bigoplus_{y'=0}^{4} S_3\left[2,y',0\right] \oplus \sum_{y'=0}^{4} S_3\left[4,y',7\right]$$

A.4 Double round sharings

A.4.1 $5 \rightarrow 10$

$$
\begin{aligned}
S'_1[x] &\leftarrow S_1[x] \oplus (S_1[x+1] \oplus 1)S_1[x+2] \oplus S_1[x+1]S_2[x+2] \oplus S_2[x+1]S_1[x+2] \\
S'_2[x] &\leftarrow S_2[x] \oplus (S_2[x+1] \oplus 1)S_2[x+2] \oplus S_2[x+1]S_3[x+2] \oplus S_3[x+1]S_2[x+2] \\
S'_3[x] &\leftarrow S_3[x] \oplus (S_3[x+1] \oplus 1)S_3[x+2] \oplus S_3[x+1]S_4[x+2] \oplus S_4[x+1]S_3[x+2] \\
S'_4[x] &\leftarrow S_4[x] \oplus (S_4[x+1] \oplus 1)S_4[x+2] \oplus S_4[x+1]S_5[x+2] \oplus S_5[x+1]S_4[x+2] \\
S'_5[x] &\leftarrow S_5[x] \oplus (S_5[x+1] \oplus 1)S_5[x+2] \oplus S_5[x+1]S_1[x+2] \oplus S_1[x+1]S_5[x+2] \\
S'_6[x] &\leftarrow S_1[x+1]S_3[x+2] \oplus S_3[x+1]S_1[x+2] \\
S'_7[x] &\leftarrow S_3[x+1]S_5[x+2] \oplus S_5[x+1]S_3[x+2] \\
S'_8[x] &\leftarrow S_5[x+1]S_2[x+2] \oplus S_2[x+1]S_5[x+2] \\
S'_9[x] &\leftarrow S_2[x+1]S_4[x+2] \oplus S_4[x+1]S_2[x+2] \\
S'_{10}[x] &\leftarrow S_4[x+1]S_1[x+2] \oplus S_1[x+1]S_4[x+2]
\end{aligned}
\tag{A.3}
$$

A.4.2 $10 \rightarrow 5$

$$
\begin{aligned}
S'_1[x] \leftarrow\ & S_1[x] \oplus S_1[x+2] \oplus S_2[x] \oplus S_2[x+2] \oplus S_1[x+1]S_2[x+2] \oplus S_1[x+1]S_3[x+2] \oplus \\
& S_1[x+1]S_1[x+2] \oplus S_1[x+1]S_6[x+2] \oplus S_1[x+1]S_9[x+2] \oplus S_1[x+1]S_{10}[x+2] \oplus \\
& S_2[x+1]S_6[x+2] \oplus S_2[x+1]S_9[x+2] \oplus S_2[x+1]S_2[x+2] \oplus S_2[x+1]S_{10}[x+2] \oplus \\
& S_{10}[x+1]S_2[x+2] \oplus S_3[x+1]S_6[x+2] \oplus S_3[x+1]S_9[x+2] \oplus S_3[x+1]S_{10}[x+2] \oplus \\
& S_6[x+1]S_9[x+2] \oplus S_9[x+1]S_6[x+2] \oplus S_6[x+1]S_{10}[x+2] \oplus S_9[x+1]S_{10}[x+2] \oplus \\
& S_2[x+1]S_3[x+2] \oplus S_3[x+1]S_1[x+2] \\
S'_2[x] \leftarrow\ & S_5[x] \oplus S_5[x+2] \oplus S_6[x] \oplus S_6[x+2] \oplus S_2[x+1]S_1[x+2] \oplus S_1[x+1]S_5[x+2] \oplus \\
& S_1[x+1]S_7[x+2] \oplus S_7[x+1]S_1[x+2] \oplus S_1[x+1]S_8[x+2] \oplus S_2[x+1]S_5[x+2] \oplus \\
& S_6[x+1]S_2[x+2] \oplus S_2[x+1]S_7[x+2] \oplus S_2[x+1]S_8[x+2] \oplus S_5[x+1]S_6[x+2] \oplus \\
& S_5[x+1]S_5[x+2] \oplus S_6[x+1]S_6[x+2] \oplus S_6[x+1]S_7[x+2] \oplus S_6[x+1]S_8[x+2] \oplus \\
& S_8[x+1]S_6[x+2] \oplus S_7[x+1]S_8[x+2] \oplus S_5[x+1]S_8[x+2] \oplus S_5[x+1]S_7[x+2] \oplus \\
& S_5[x+1]S_2[x+2] \oplus S_6[x+1]S_1[x+2] \qquad \text{(A.4)} \\
S'_3[x] \leftarrow\ & S_4[x] \oplus S_4[x+2] \oplus S_8[x] \oplus S_8[x+2] \oplus S_4[x+1]S_4[x+2] \oplus S_8[x+1]S_8[x+2] \oplus \\
& S_4[x+1]S_1[x+2] \oplus S_5[x+1]S_1[x+2] \oplus S_8[x+1]S_1[x+2] \oplus S_9[x+1]S_1[x+2] \oplus \\
& S_4[x+1]S_5[x+2] \oplus S_4[x+1]S_8[x+2] \oplus S_4[x+1]S_9[x+2] \oplus S_9[x+1]S_4[x+2] \oplus \\
& S_4[x+1]S_{10}[x+2] \oplus S_8[x+1]S_5[x+2] \oplus S_5[x+1]S_9[x+2] \oplus S_9[x+1]S_5[x+2] \oplus \\
& S_5[x+1]S_{10}[x+2] \oplus S_8[x+1]S_9[x+2] \oplus S_8[x+1]S_{10}[x+2] \oplus S_{10}[x+1]S_8[x+2] \oplus \\
& S_{10}[x+1]S_9[x+2] \oplus S_{10}[x+1]S_1[x+2] \oplus S_1[x+1]S_4[x+2] \\
S'_4[x] \leftarrow\ & S_3[x] \oplus S_3[x+2] \oplus S_{10}[x] \oplus S_{10}[x+2] \oplus S_3[x+1]S_3[x+2] \oplus S_{10}[x+1]S_{10}[x+2] \oplus \\
& S_3[x+1]S_5[x+2] \oplus S_5[x+1]S_3[x+2] \oplus S_6[x+1]S_3[x+2] \oplus S_3[x+1]S_7[x+2] \oplus \\
& S_5[x+1]S_4[x+2] \oplus S_4[x+1]S_6[x+2] \oplus S_6[x+1]S_4[x+2] \oplus S_4[x+1]S_7[x+2] \oplus \\
& S_{10}[x+1]S_4[x+2] \oplus S_6[x+1]S_5[x+2] \oplus S_7[x+1]S_5[x+2] \oplus S_{10}[x+1]S_5[x+2] \oplus \\
& S_7[x+1]S_6[x+2] \oplus S_{10}[x+1]S_6[x+2] \oplus S_7[x+1]S_{10}[x+2] \oplus S_{10}[x+1]S_7[x+2] \oplus \\
& S_{10}[x+1]S_3[x+2] \oplus S_3[x+1]S_4[x+2] \\
S'_5[x] \leftarrow\ & S_7[x] \oplus S_7[x+2] \oplus S_9[x] \oplus S_9[x+2] \oplus S_7[x+1]S_7[x+2] \oplus S_9[x+1]S_9[x+2] \oplus \\
& S_2[x+1]S_4[x+2] \oplus S_4[x+1]S_2[x+2] \oplus S_7[x+1]S_2[x+2] \oplus S_8[x+1]S_2[x+2] \oplus \\
& S_4[x+1]S_3[x+2] \oplus S_7[x+1]S_3[x+2] \oplus S_8[x+1]S_3[x+2] \oplus S_9[x+1]S_3[x+2] \oplus \\
& S_7[x+1]S_4[x+2] \oplus S_8[x+1]S_4[x+2] \oplus S_8[x+1]S_7[x+2] \oplus S_9[x+1]S_7[x+2] \oplus \\
& S_9[x+1]S_8[x+2] \oplus S_7[x+1]S_9[x+2] \oplus S_3[x+1]S_8[x+2] \oplus S_9[x+1]S_2[x+2] \\
& S_3[x+1]S_2[x+2]
\end{aligned}
$$

A.4.3 First layer $6 \to 6$

$$
\begin{aligned}
S'_1[x] \quad &\leftarrow \quad S_2[x] \oplus (S_2[x+1] \oplus 1) S_2[x+2] \oplus S_1[x+1] S_2[x+2] \oplus S_2[x+1] S_1[x+2] \oplus \\
&\qquad S_1[x+1] S_3[x+2] \oplus S_3[x+1] S_1[x+2] \oplus S_2[x+1] S_3[x+2] \oplus S_3[x+1] S_2[x+2] \\
S'_2[x] \quad &\leftarrow \quad S_5[x] \oplus (S_5[x+1] \oplus 1) S_5[x+2] \oplus S_1[x+1] S_4[x+2] \oplus S_4[x+1] S_1[x+2] \oplus \\
&\qquad S_1[x+1] S_5[x+2] \oplus S_5[x+1] S_1[x+2] \oplus S_4[x+1] S_5[x+2] \oplus S_5[x+1] S_4[x+2] \\
S'_3[x] \quad &\leftarrow \quad S_1[x] \oplus (S_1[x+1] \oplus 1) S_1[x+2] \oplus S_1[x+1] S_6[x+2] \oplus S_6[x+1] S_1[x+2] \oplus \\
&\qquad S_4[x+1] S_6[x+2] \oplus S_6[x+1] S_4[x+2] \qquad \text{(A.5)} \\
S'_4[x] \quad &\leftarrow \quad S_6[x] \oplus (S_6[x+1] \oplus 1) S_6[x+2] \oplus S_2[x+1] S_5[x+2] \oplus S_5[x+1] S_2[x+2] \oplus \\
&\qquad S_2[x+1] S_6[x+2] \oplus S_6[x+1] S_2[x+2] \oplus S_5[x+1] S_6[x+2] \oplus S_6[x+1] S_5[x+2] \\
S'_5[x] \quad &\leftarrow \quad S_3[x] \oplus (S_3[x+1] \oplus 1) S_3[x+2] \oplus S_3[x+1] S_5[x+2] \oplus S_5[x+1] S_3[x+2] \oplus \\
&\qquad S_3[x+1] S_6[x+2] \oplus S_6[x+1] S_3[x+2] \\
S'_6[x] \quad &\leftarrow \quad S_4[x] \oplus (S_4[x+1] \oplus 1) S_4[x+2] \oplus S_2[x+1] S_4[x+2] \oplus S_4[x+1] S_2[x+2] \oplus \\
&\qquad S_3[x+1] S_4[x+2] \oplus S_4[x+1] S_3[x+2]
\end{aligned}
$$

A.4.4 Second layer $6 \to 6$

$$
\begin{aligned}
S'_1[x] &\leftarrow S_6[x] \oplus (S_6[x+1] \oplus 1)S_6[x+2] \oplus S_5[x+1]S_6[x+2] \oplus S_6[x+1]S_5[x+2] \oplus \\
&\quad S_6[x+1]S_4[x+2] \oplus S_4[x+1]S_6[x+2] \\
S'_2[x] &\leftarrow S_2[x] \oplus (S_2[x+1] \oplus 1)S_2[x+2] \oplus S_2[x+1]S_3[x+2] \oplus S_3[x+1]S_2[x+2] \oplus \\
&\quad S_2[x+1]S_5[x+2] \oplus S_5[x+1]S_2[x+2] \oplus S_3[x+1]S_5[x+2] \oplus S_5[x+1]S_3[x+2] \\
S'_3[x] &\leftarrow S_4[x] \oplus (S_4[x+1] \oplus 1)S_4[x+2] \oplus S_2[x+1]S_4[x+2] \oplus S_4[x+1]S_2[x+2] \oplus \\
&\quad S_3[x+1]S_4[x+2] \oplus S_4[x+1]S_3[x+2] \qquad \text{(A.6)} \\
S'_4[x] &\leftarrow S_5[x] \oplus (S_5[x+1] \oplus 1)S_5[x+2] \oplus S_1[x+1]S_4[x+2] \oplus S_4[x+1]S_1[x+2] \oplus \\
&\quad S_1[x+1]S_5[x+2] \oplus S_5[x+1]S_1[x+2] \oplus S_4[x+1]S_5[x+2] \oplus S_5[x+1]S_4[x+2] \\
S'_5[x] &\leftarrow S_3[x] \oplus (S_3[x+1] \oplus 1)S_3[x+2] \oplus S_3[x+1]S_1[x+2] \oplus S_1[x+1]S_3[x+2] \oplus \\
&\quad S_3[x+1]S_6[x+2] \oplus S_6[x+1]S_3[x+2] \\
S'_6[x] &\leftarrow S_1[x] \oplus (S_1[x+1] \oplus 1)S_1[x+2] \oplus S_1[x+1]S_2[x+2] \oplus S_2[x+1]S_1[x+2] \oplus \\
&\quad S_1[x+1]S_6[x+2] \oplus S_6[x+1]S_1[x+2] \oplus S_2[x+1]S_6[x+2] \oplus S_6[x+1]S_2[x+2]
\end{aligned}
$$

Bibliography

[1] SAKURA hardware security project.

[2] Draft FIPS 202, SHA-3 standard: Permutation-based hash and extendable-output functions. In *NIST Publication, US Dept. of Commerce* (August 2015).

[3] Abramovici, M., Breuer, M., and Friedman, A. Digital systems testing and testable design. Wiley-IEEE Press, September 1994.

[4] Agarwal, V. K., and Fung, A. S. F. Multiple fault testing of large circuits by single fault test sets. *IEEE Trans. Computers 30*, 11 (1981), 855–865.

[5] Aghaie, A., Moradi, A., Rasoolzadeh, S., Schellenberg, F., and Schneider, T. Impeccable circuits. *IACR Cryptology ePrint Archive 2018* (2018), 203.

[6] Amiel, F., Villegas, K., Feix, B., and Marcel, L. Passive and active combined attacks: Combining fault attacks and side channel analysis. In *FDTC 2007* (2007), pp. 92–102.

[7] Arribas, V. VerMFi: Verification tool for masked implementations and fault injection.

[8] Arribas, V. Beyond the limits: SHA-3 in just 49 slices. In *FPL* (2019), IEEE, pp. 239–245.

[9] Arribas, V., Bilgin, B., Petrides, G., Nikova, S., and Rijmen, V. Rhythmic Keccak: SCA security and low latency in hw. *IACR Transactions on Cryptographic Hardware and Embedded Systems 2018*, 1 (2018), 269–290.

[10] Arribas, V., Cnudde, T. D., and Sijacic, D. Glitch-resistant masking schemes as countermeasure against fault sensitivity analysis. In *FDTC* (2018), IEEE Computer Society, pp. 27–34.

[11] Arribas, V., Nikova, S., and Rijmen, V. Guards in action: First-order SCA secure implementations of ketje without additional randomness. In *21st Euromicro Conference on Digital System Design, DSD 2018, Prague, Czech Republic, August 29-31, 2018* (2018), pp. 492–499.

[12] Arribas, V., Nikova, S., and Rijmen, V. VerMI: Verification Tool for Masked Implementations. In *25th IEEE International Conference on Electronics, Circuits, and Systems* (Bordeaux,FR, 2018), IEEE, p. 4.

[13] Arribas, V., Nikova, S., and Rijmen, V. Guards in action: First-order SCA secure implementations of KETJE without additional randomness. *Microprocessors and Microsystems - Embedded Hardware Design 71* (2019).

[14] Arribas, V., Wegener, F., Moradi, A., and Nikova, S. Cryptographic fault diagnosis using verfi. *IACR Cryptology ePrint Archive 2019* (2019), 1312.

[15] Banik, S., Bogdanov, A., Isobe, T., Shibutani, K., Hiwatari, H., Akishita, T., and Regazzoni, F. Midori: A block cipher for low energy. In *ASIACRYPT 2015* (2015), vol. 9453 of *Lecture Notes in Computer Science*, Springer, pp. 411–436.

[16] Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., and Whelan, C. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE 94*, 2 (2006), 370–382.

[17] Barthe, G., Belaïd, S., Cassiers, G., Fouque, P., Grégoire, B., and Standaert, F. maskverif: Automated verification of higher-order masking in presence of physical defaults. In *ESORICS (1)* (2019), vol. 11735 of *Lecture Notes in Computer Science*, Springer, pp. 300–318.

[18] Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P., Grégoire, B., and Strub, P. *Verified Proofs of Higher-Order Masking.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 457–485.

[19] Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P., Grégoire, B., Strub, P., and Zucchini, R. Strong non-interference and type-directed higher-order masking. In *ACM Conference on Computer and Communications Security* (2016), ACM, pp. 116–129.

[20] Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P.-A., Grégoire, B., Standaert, F.-X., and Strub, P.-Y. Improved parallel mask refreshing algorithms - generic solutions with parametrized non-interference & automated optimizations. *IACR Cryptology ePrint Archive 2018* (2018), 505.

[21] Barzilai, Z., Carter, J. L., Rosen, B. K., and Rutledge, J. D. HSS–A high-speed simulator. *IEEE Trans. on CAD of Integrated Circuits and Systems 6*, 4 (1987), 601–617.

[22] Battistello, A., and Giraud, C. Fault analysis of infective AES computations. In *FDTC 2013* (2013), pp. 101–107.

[23] Bayrak, A. G., Regazzoni, F., Novo, D., and Ienne, P. *Sleuth: Automated Verification of Software Power Analysis Countermeasures.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 293–310.

[24] Beaver, D. Precomputing oblivious transfer. In *CRYPTO* (1995), vol. 963 of *Lecture Notes in Computer Science*, Springer, pp. 97–109.

[25] Becker, G., Cooper, J., De Mulder, E., Goodwill, G., Jaffe, J., Kenworthy, G., Kouzminov, T., Leiserson, A., Marson, M., Rohatgi, P., et al. Test vector leakage assessment (tvla) methodology in practice. In *International Cryptographic Module Conference* (2013), vol. 1001, p. 13.

[26] Beierle, C., Leander, G., Moradi, A., and Rasoolzadeh, S. CRAFT: lightweight tweakable block cipher with efficient protection against DFA attacks. *IACR Trans. Symmetric Cryptol. 2019*, 1 (2019), 5–45.

[27] Belaïd, S., Benhamouda, F., Passelègue, A., Prouff, E., Thillard, A., and Vergnaud, D. Randomness complexity of private circuits for multiplication. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II* (2016), pp. 616–648.

[28] Ben-Or, M., Goldwasser, S., and Wigderson, A. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC* (1988), ACM, pp. 1–10.

[29] Bendlin, R., Damgård, I., Orlandi, C., and Zakarias, S. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT* (2011), vol. 6632 of *Lecture Notes in Computer Science*, Springer, pp. 169–188.

[30] Bertoni, G., Breveglieri, L., Koren, I., Maistri, P., and Piuri, V. Error analysis and detection procedures for a hardware implementation of the advanced encryption standard. *IEEE Trans. Computers 52*, 4 (2003), 492–505.

[31] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. Building power analysis resistant implementations of Keccak. Second SHA-3 candidate conference, August 2010.

[32] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. The keccak reference. http://http://keccak.noekeon.org/, January 2010.

[33] Bertoni, G., Daemen, J., Peeters, M., Assche, G. V., and Keer, R. V. 1001 ways to implement Keccak. In *Third SHA-3 candidate conference* (2012).

[34] Bertoni, G., Daemen, J., Peeters, M., Assche, G. V., and Keer, R. V. Keccak implementation overview. In *Report, STMicroelectronics, Antwerp, Belgium* (2012).

[35] Bertoni, G., Daemen, J., Peeters, M., Assche, G. V., and Keer, R. V. Caesar submission: Ketje v2, September 2016.

[36] Bertoni, G., and Martinoli, M. A methodology for the characterisation of leakages in combinatorial logic. In *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings* (2016), pp. 363–382.

[37] Biham, E., and Shamir, A. Differential fault analysis of secret key cryptosystems. In *CRYPTO* (1997), vol. 1294 of *Lecture Notes in Computer Science*, Springer, pp. 513–525.

[38] Bilgin, B., Daemen, J., Nikov, V., Nikova, S., Rijmen, V., and Assche, G. V. Efficient and first-order DPA resistant implementations of Keccak. in CARDIS, volume 8419 of LNCS pp 187-199, June 2014.

[39] Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., and Rijmen, V. A More Efficient AES Threshold Implementation. In *Progress in Cryptology-AFRICACRYPT 2014*, D. Pointcheval and D. Vergnaud, Eds., vol. 8469 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 267–284.

[40] Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., and Rijmen, V. Higher-order threshold implementations. In *ASIACRYPT (2)* (2014), vol. 8874 of *Lecture Notes in Computer Science*, Springer, pp. 326–343.

[41] Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., and Rijmen, V. Trade-Offs for Threshold Implementations Illustrated on AES. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 34*, 7 (July 2015), 1188–1200.

[42] Black, J. R. Electromigration—a brief survey and some recent results. *IEEE Transactions on Electron Devices 16*, 4 (April 1969), 338–347.

[43] Bloem, R., Gross, H., Iusupov, R., Könighofer, B., Mangard, S., and Winter, J. Formal Verification of Masked Hardware Implementations in the Presence of Glitches. Cryptology ePrint Archive, Report 2017/897, 2017. `http://eprint.iacr.org/2017/897`.

[44] Boneh, D., DeMillo, R. A., and Lipton, R. J. On the importance of checking cryptographic protocols for faults (extended abstract). In *EUROCRYPT* (1997), vol. 1233 of *Lecture Notes in Computer Science*, Springer, pp. 37–51.

[45] Boneh, D., DeMillo, R. A., and Lipton, R. J. On the importance of eliminating errors in cryptographic computations. *J. Cryptology 14*, 2 (2001), 101–119.

[46] Bosio, A., and Natale, G. D. LIFTING: A flexible open-source fault simulator. In *17th IEEE Asian Test Symposium, ATS 2008, Sapporo, Japan, November 24-27, 2008* (2008), pp. 35–40.

[47] Brier, E., Clavier, C., and Olivier, F. Correlation power analysis with a leakage model. In *CHES* (2004), vol. 3156 of *Lecture Notes in Computer Science*, Springer, pp. 16–29.

[48] Bringer, J., Carlet, C., Chabanne, H., Guilley, S., and Maghrebi, H. Orthogonal direct sum masking - A smartcard friendly computation paradigm in a code, with builtin protection against side-channel and fault attacks. In *WISTP 2014. Proceedings* (2014), pp. 40–56.

[49] Bryant, R. E., Beatty, D. L., Brace, K. S., Cho, K., and Sheffler, T. J. COSMOS: A compiled simulator for MOS circuits. In *DAC* (1987), IEEE Computer Society Press / ACM, pp. 9–16.

[50] Bulck, J. V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium* (2018), USENIX Association, pp. 991–1008.

[51] Bulck, J. V., Oswald, D., Marin, E., Aldoseri, A., Garcia, F. D., and Piessens, F. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *CCS* (2019), ACM, pp. 1741–1758.

[52] Bushnell, M., and Agrawal, V. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer Publishing Company, Incorporated, 2013.

[53] Butler, K. M., Kapur, R., Mercer, M. R., and Ross, D. E. The roles of controllability and observability in design for test. In *10th IEEE VLSI Test Symposium (VTS'92), 7-9 Apr 1992, Atlantic City, NJ, USA* (1992), pp. 211–216.

[54] Camurati, P., Prinetto, P., Rebaudengo, M., and Reorda, M. S. Improved techniques for multiple stuck-at fault analysis using single stuck-at fault test sets. In *[Proceedings] 1992 IEEE International Symposium on Circuits and Systems*, IEEE.

[55] Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Bulck, J. V., and Yarom, Y. Fallout: Leaking data on meltdown-resistant cpus. In *CCS* (2019), ACM, pp. 769–784.

[56] Cannière, C. D., Dunkelman, O., and Knezevic, M. KATAN and KTANTAN - A family of small and efficient hardware-oriented block ciphers. In *CHES* (2009), vol. 5747 of *Lecture Notes in Computer Science*, Springer, pp. 272–288.

[57] Canright, D. A very compact s-box for AES. In *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings* (2005), pp. 441–455.

[58] Chari, S., Rao, J. R., and Rohatgi, P. Template attacks. In *CHES* (2002), vol. 2523 of *Lecture Notes in Computer Science*, Springer, pp. 13–28.

[59] Chatterjee, D., DeOrio, A., and Bertacco, V. GCS: high-performance gate-level simulation with gpgpus. In *DATE* (2009), IEEE, pp. 1332–1337.

[60] Clavier, C. Secret external encodings do not prevent transient fault analysis. In *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings* (2007), pp. 181–194.

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

[62] Cnudde, T. D., Bilgin, B., Gierlichs, B., Nikov, V., Nikova, S., and Rijmen, V. Does coupling affect the security of masked implementations? In *COSADE* (2017), vol. 10348 of *Lecture Notes in Computer Science*, Springer, pp. 1–18.

[63] Cnudde, T. D., Bilgin, B., Reparaz, O., and Nikova, S. Higher-order glitch resistant implementation of the PRESENT s-box. In *BalkanCryptSec* (2014), vol. 9024 of *Lecture Notes in Computer Science*, Springer, pp. 75–93.

[64] Cnudde, T. D., and Nikova, S. More efficient private circuits II through threshold implementations. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016* (2016), pp. 114–124.

[65] Cnudde, T. D., Reparaz, O., Bilgin, B., Nikova, S., Nikov, V., and Rijmen, V. Masking AES with d+1 shares in hardware. In *CHES* (2016), vol. 9813 of *Lecture Notes in Computer Science*, Springer, pp. 194–212.

[66] Cooper, J., DeMulder, E., Goodwill, G., Jaffe, J., Kenworthy, G., and Rohatgi, P. Test Vector Leakage Assessment (TVLA) Methodology in Practice. International Cryptographic Module Conference, 2013. `http://icmc-2013.org/wp/wp-content/uploads/2013/09/goodwillkenworthtestvector.pdf`.

[67] Coron, J. Higher order masking of look-up tables. In *EUROCRYPT* (2014), vol. 8441 of *Lecture Notes in Computer Science*, Springer, pp. 441–458.

[68] Coron, J.-S. Formal Verification of Side-channel Countermeasures via Elementary Circuit Transformations. Cryptology ePrint Archive, Report 2017/879, 2017. `http://eprint.iacr.org/2017/879`.

[69] Daemen, J. Changing of the guards: A simple and efficient method for achieving uniformity in threshold sharing. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings* (2017), pp. 137–153.

[70] Daemen, J., Dobraunig, C., Eichlseder, M., Gross, H., Mendel, F., and Primas, R. Protecting against statistical ineffective fault attacks. *IACR Cryptology ePrint Archive 2019* (2019), 536.

[71] Daemen, J., and Rijmen, V. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[72] Damgård, I., Pastro, V., Smart, N. P., and Zakarias, S. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO* (2012), vol. 7417 of *Lecture Notes in Computer Science*, Springer, pp. 643–662.

[73] Dassance, F., and Venelli, A. Combined fault and side-channel attacks on the aes key schedule. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography* (Sept 2012), pp. 63–71.

[74] De Meyer, L., Arribas, V., Nikova, S., Nikov, V., and Rijmen, V. M&m: Masks and macs against physical attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst. 2019*, 1 (2019), 25–50.

[75] Dobraunig, C., Eichlseder, M., Gross, H., Mangard, S., Mendel, F., and Primas, R. Statistical ineffective fault attacks on masked AES with fault countermeasures. *IACR Cryptology ePrint Archive 2018* (2018), 357.

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

[77] Dobraunig, C., Eichlseder, M., Korak, T., Mangard, S., Mendel, F., and Primas, R. Sifa: Exploiting ineffective fault inductions on symmetric cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems 2018*, 3 (Aug. 2018), 547–572.

[78] Duc, A., Faust, S., and Standaert, F. Making masking security proofs concrete - or how to evaluate the security of any leaking device. In *EUROCRYPT (1)* (2015), vol. 9056 of *Lecture Notes in Computer Science*, Springer, pp. 401–429.

[79] Duc, A., Faust, S., and Standaert, F.-X. *Making Masking Security Proofs Concrete*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 401–429.

[80] Dziembowski, S., and Pietrzak, K. Leakage-resilient cryptography. In *FOCS* (2008), IEEE Computer Society, pp. 293–302.

[81] Eldib, H., Wang, C., and Schaumont, P. *SMT-Based Verification of Software Countermeasures against Side-Channel Attacks*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 62–77.

[82] Eldib, H., Wang, C., Taha, M., and Schaumont, P. Qms: Evaluating the side-channel resistance of masked software from source code. In *Proceedings of the 51st Annual Design Automation Conference* (New York, NY, USA, 2014), DAC '14, ACM, pp. 209:1–209:6.

[83] Endo, S., Li, Y., Homma, N., Sakiyama, K., Ohta, K., and Aoki, T. An efficient countermeasure against fault sensitivity analysis using configurable delay blocks. In *FDTC* (2012), IEEE Computer Society, pp. 95–102.

[84] Endo, S., Li, Y., Homma, N., Sakiyama, K., Ohta, K., Fujimoto, D., Nagata, M., Katashita, T., Danger, J., and Aoki, T. A silicon-level countermeasure against fault sensitivity analysis and its evaluation. *IEEE Trans. VLSI Syst. 23*, 8 (2015), 1429–1438.

[85] Endo, S., Sugawara, T., Homma, N., Aoki, T., and Satoh, A. An on-chip glitchy-clock generator for testing fault injection attacks. *J. Cryptographic Engineering 1*, 4 (2011), 265–270.

[86] Endo, S., Sugawara, T., Homma, N., Aoki, T., and Satoh, A. A configurable on-chip glitchy-clock generator for fault injection experiments. *IEICE Transactions 95-A*, 1 (2012), 263–266.

[87] Fan, X., Moore, W. R., Hora, C., and Gronthoud, G. Stuck-open fault diagnosis with stuck-at model. In *10th European Test Symposium, ETS 2005, Tallinn, Estonia, May 22-25, 2005* (2005), pp. 182–187.

[88] Fan, X., Moore, W. R., Hora, C., and Gronthoud, G. Extending gate-level diagnosis tools to CMOS intra-gate faults. *IET Computers & Digital Techniques 1*, 6 (2007), 685–693.

[89] Faust, S., Grosso, V., Merino Del Pozo, S., Paglialonga, C., and Standaert, F.-X. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Transactions on Cryptographic Hardware and Embedded Systems 2018*, 3 (Aug. 2018), 89–120.

[90] Faust, S., Grosso, V., Pozo, S. M. D., Paglialonga, C., and Standaert, F. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Trans. Cryptogr. Hardw. Embed. Syst. 2018*, 3 (2018), 89–120.

[91] Gennaro, R., Lysyanskaya, A., Malkin, T., Micali, S., and Rabin, T. Algorithmic tamper-proof (ATP) security: Theoretical foundations for security against hardware tampering. In *TCC* (2004), vol. 2951 of *Lecture Notes in Computer Science*, Springer, pp. 258–277.

[92] Ghalaty, N. F., Aysu, A., and Schaumont, P. Analyzing and eliminating the causes of fault sensitivity analysis. In *DATE* (2014), European Design and Automation Association, pp. 1–6.

[93] Gierlichs, B., Schmidt, J., and Tunstall, M. Infective computation and dummy rounds: Fault protection for block ciphers without check-before-output. In *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings* (2012), pp. 305–321.

[94] Gierlichs, O. R. B., and Verbauwhede, I. Fast leakage assessment. In *Cryptographic Hardware and Embedded Systems – CHES 2017* (Cham, 2017), W. Fischer and N. Homma, Eds., Springer International Publishing, pp. 387–399.

[95] Giraud, C. DFA on AES. In *Advanced Encryption Standard - AES, 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited Papers* (2004), pp. 27–41.

[96] Gross, H., Iusupov, R., and Bloem, R. Generic low-latency masking in hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst. 2018*, 2 (2018), 1–21.

[97] Gross, H., and Mangard, S. Reconciling d+1 masking in hardware and software. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings* (2017), pp. 115–136.

[98] Gross, H., Mangard, S., and Korak, T. An efficient side-channel protected AES implementation with arbitrary protection order. In *Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14-17, 2017, Proceedings* (2017), pp. 95–112.

[99] Gross, H., Schaffenrath, D., and Mangard, S. Higher-order side-channel protected implementations of keccak. Cryptology ePrint Archive, Report 2017/395, 2017. `https://eprint.iacr.org/2017/395`.

[100] Gross, H., Schaffenrath, D., and Mangard, S. Higher-Order Side-Channel Protected Implementations of KECCAK. In *2017 Euromicro Conference on Digital System Design (DSD)* (Aug 2017), pp. 205–212.

[101] Grosso, V., Prouff, E., and Standaert, F. Efficient masked s-boxes processing - A step forward -. In *Progress in Cryptology - AFRICACRYPT 2014 - 7th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 28-30, 2014. Proceedings* (2014), pp. 251–266.

[102] Guo, J., Peyrin, T., Poschmann, A., and Robshaw, M. J. B. The LED block cipher. In *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings* (2011), pp. 326–341.

[103] Hansen, C. Hardware logic simulation by compilation. In *DAC* (1988), ACM, pp. 712–716.

[104] Hughes, J. L. A., and McCluskey, E. J. Multiple stuck-at fault coverage of single stuck-at fault test sets. In *Proceedings International Test Conference 1986, Washington, D.C., USA, September 1986* (1986), IEEE Computer Society, pp. 368–374.

[105] Inc., V. S. 0.18 μm VIP Standard cell library tapeout ready, partnumber: UMCL18G212T3, process: UMC logic 0.18μm generic II technology: 0.18μm, July 2004.

[106] Ioannou, L., Michail, H. E., and Voyiatzis, A. G. High performance pipelined FPGA implementation of the SHA-3 hash algorithm. In *2015 4th Mediterranean Conference on Embedded Computing (MECO)* (June 2015), pp. 68–71.

[107] Ishai, Y., Prabhakaran, M., Sahai, A., and Wagner, D. A. Private circuits II: keeping secrets in tamperable circuits. In *EUROCRYPT 2006* (2006), vol. 4004 of *Lecture Notes in Computer Science*, Springer, pp. 308–327.

[108] Ishai, Y., Sahai, A., and Wagner, D. Private circuits: Securing hardware against probing attacks. In *CRYPTO* (2003), vol. 2729 of *Lecture Notes in Computer Science*, Springer, pp. 463–481.

[109] Jha, N. K. Detecting multiple faults in CMOS circuits. In *Proceedings International Test Conference 1986, Washington, D.C., USA, September 1986* (1986), IEEE Computer Society, pp. 514–519.

[110] Jungk, B. *FPGA-based evaluation of cryptographic algorithms.* PhD thesis, Goethe University Frankfurt am Main, 2016.

[111] Jungk, B., and Apfelbeck, J. Area-efficient FPGA implementations of the SHA-3 finalists. In *2011 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2011, Cancun, Mexico, November 30 - December 2, 2011* (2011), pp. 235–241.

[112] Jungk, B., and Stöttinger, M. Among slow dwarfs and fast giants: A systematic design space exploration of KECCAK. In *2013 8th International Workshop on Reconfigurable and Communication-Centric*

Systems-on-Chip (ReCoSoC), Darmstadt, Germany, July 10-12, 2013 (2013), pp. 1–8.

[113] Jungk, B., and Stöttinger, M. Hobbit - smaller but faster than a dwarf: Revisiting lightweight SHA-3 FPGA implementations. In *International Conference on ReConFigurable Computing and FPGAs, ReConFig 2016, Cancun, Mexico, November 30 - Dec. 2, 2016* (2016), pp. 1–7.

[114] Karaklajić, D., Schmidt, J.-M., and Verbauwhede, I. Hardware designer's guide to fault attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 21*, 12 (Dec 2013), 2295–2306.

[115] Karpovsky, M. G., Kulikowski, K. J., and Taubin, A. Differential fault analysis attack resistant architectures for the advanced encryption standard. In *CARDIS 2004, 22-27 August 2004, Toulouse, France* (2004), pp. 177–192.

[116] Karri, R., Kuznetsov, G., and Gössel, M. Parity-based concurrent error detection of substitution-permutation network block ciphers. In *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings* (2003), pp. 113–124.

[117] Keller, M., Orsini, E., and Scholl, P. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *ACM Conference on Computer and Communications Security* (2016), ACM, pp. 830–842.

[118] Kerckhof, S., Durvaux, F., Veyrat-Charvillon, N., Regazzoni, F., de Dormale, G. M., and Standaert, F. Compact FPGA implementations of the five SHA-3 finalists. In *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers* (2011), pp. 217–233.

[119] Kerckhoffs, A. *La cryptographie militaire.* J. Sci. Mil., 1883.

[120] Khanna, P., Rebeiro, C., and Hazra, A. XFC: A framework for exploitable fault characterization in block ciphers. In *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017* (2017), pp. 8:1–8:6.

[121] Knudsen, L. R., and Miolane, C. V. Counting equations in algebraic attacks on block ciphers. *Int. J. Inf. Sec. 9*, 2 (2010), 127–135.

[122] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy* (2019), IEEE, pp. 1–19.

[123] Kocher, P. C. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO* (1996), vol. 1109 of *Lecture Notes in Computer Science*, Springer, pp. 104–113.

[124] Kocher, P. C., Jaffe, J., and Jun, B. Differential Power Analysis. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings* (1999), pp. 388–397.

[125] Lee, H. K., and Ha, D. S. HOPE: an efficient parallel fault simulator for synchronous sequential circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems 15*, 9 (1996), 1048–1058.

[126] Lee, Y. S., and Maurer, P. M. Two new techniques for compiled multi-delay logic simulation. In *DAC* (1992), IEEE Computer Society Press, pp. 420–423.

[127] Lewis, D. M. A hierarchical compiled code event-driven logic simulator. *IEEE Trans. on CAD of Integrated Circuits and Systems 10*, 6 (1991), 726–737.

[128] Li, Y., Ohta, K., and Sakiyama, K. Toward effective countermeasures against an improved fault sensitivity analysis. *IEICE Transactions 95-A*, 1 (2012), 234–241.

[129] Li, Y., Sakiyama, K., Gomisawa, S., Fukunaga, T., Takahashi, J., and Ohta, K. Fault sensitivity analysis. In *CHES* (2010), vol. 6225 of *Lecture Notes in Computer Science*, Springer, pp. 320–334.

[130] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium* (2018), USENIX Association, pp. 973–990.

[131] Lomné, V., Roche, T., and Thillard, A. On the need of randomness in fault attack countermeasures - application to AES. In *FDTC 2012* (2012), pp. 85–94.

[132] MacWilliams, F., and Sloane, N. *The Theory of Error-Correcting Codes*, 2nd ed. North-holland Publishing Company, 1978.

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

[134] Maurer, P. M. Event driven simulation without loops or conditionals. In *ICCAD* (2000), IEEE Computer Society, pp. 23–26.

[135] Melzani, F., and Palomba, A. Enhancing fault sensitivity analysis through templates. In *HOST* (2013), IEEE Computer Society, pp. 25–28.

[136] Mischke, O., Moradi, A., and Güneysu, T. Fault sensitivity analysis meets zero-value attack. In *FDTC* (2014), IEEE Computer Society, pp. 59–67.

[137] Moos, T., Moradi, A., Schneider, T., and Standaert, F. Glitch-resistant masking revisited or why proofs in the robust probing model are needed. *IACR Trans. Cryptogr. Hardw. Embed. Syst. 2019*, 2 (2019), 256–292.

[138] Moradi, A., and Mischke, O. On the simplicity of converting leakages from multivariate to univariate - (case study of a glitch-resistant masking scheme). In *CHES* (2013), vol. 8086 of *Lecture Notes in Computer Science*, Springer, pp. 1–20.

[139] Moradi, A., Mischke, O., and Paar, C. One attack to rule them all: Collision timing attack versus 42 AES ASIC cores. *IEEE Trans. Computers 62*, 9 (2013), 1786–1798.

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

[141] Moradi, A., Poschmann, A., Ling, S., Paar, C., and Wang, H. Pushing the limits: A very compact and a threshold implementation of AES. In *EUROCRYPT* (2011), vol. 6632 of *Lecture Notes in Computer Science*, Springer, pp. 69–88.

[142] Moss, A., Oswald, E., Page, D., and Tunstall, M. *Compiler Assisted Masking*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 58–75.

[143] Mukhopadhyay, D. An improved fault based attack of the advanced encryption standard. In *AFRICACRYPT* (2009), vol. 5580 of *Lecture Notes in Computer Science*, Springer, pp. 421–434.

[144] NANGATE. The NanGate 45nm Open Cell Library. Available at `http://www.nangate.com`.

[145] Nielsen, J. B., Nordholt, P. S., Orlandi, C., and Burra, S. S. A new approach to practical active-secure two-party computation. In *CRYPTO* (2012), vol. 7417 of *Lecture Notes in Computer Science*, Springer, pp. 681–700.

[146] Niermann, T. M., Cheng, W., and Patel, J. H. PROOFS: a fast, memory-efficient sequential circuit fault simulator. *IEEE Trans. on CAD of Integrated Circuits and Systems 11*, 2 (1992), 198–207.

[147] Nikova, S., Rechberger, C., and Rijmen, V. Threshold implementations against side-channel attacks and glitches. In *ICICS* (2006), vol. 4307 of *Lecture Notes in Computer Science*, Springer, pp. 529–545.

[148] Nikova, S., Rijmen, V., and Schläffer, M. Secure hardware implementation of non-linear functions in the presence of glitches. In *ICISC* (2008), vol. 5461 of *Lecture Notes in Computer Science*, Springer, pp. 218–234.

[149] Nikova, S., Rijmen, V., and Schläffer, M. Secure hardware implementation of nonlinear functions in the presence of glitches. *J. Cryptology 24*, 2 (2011), 292–321.

[150] Otto, M. *Fault attacks and countermeasures*. PhD thesis, University of Paderborn, Germany, 2005.

[151] Poschmann, A., Moradi, A., Khoo, K., Lim, C., Wang, H., and Ling, S. Side-channel resistant crypto for less than 2, 300 GE. *J. Cryptology 24*, 2 (2011), 322–345.

[152] Prouff, E., and Roche, T. Higher-order glitches free implementation of the AES using secure multi-party computation protocols. In *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings* (2011), pp. 63–78.

[153] Prouff, E., and Roche, T. Higher-order glitches free implementation of the AES using secure multi-party computation protocols. In *CHES* (2011), vol. 6917 of *Lecture Notes in Computer Science*, Springer, pp. 63–78.

[154] Purnal, A., Arribas, V., and DeMeyer, L. Trade-offs in protecting Keccak against combined side-channel and fault attacks. In *Constructive Side-Channel Analysis and Secure Design - 10th International Workshop, COSADE 2019* (2019).

[155] Rashid, L., Pattabiraman, K., and Gopalakrishnan, S. Characterizing the impact of intermittent hardware faults on programs. *IEEE Trans. Reliability 64*, 1 (2015), 297–310.

[156] Reparaz, O. Detecting flawed masking schemes with leakage detection tests. In *Revised Selected Papers of the 23rd International Conference on Fast Software Encryption - Volume 9783* (New York, NY, USA, 2016), FSE 2016, Springer-Verlag New York, Inc., pp. 204–222.

[157] Reparaz, O., Bilgin, B., Nikova, S., Gierlichs, B., and Verbauwhede, I. Consolidating masking schemes. In *CRYPTO (1)* (2015), vol. 9215 of *Lecture Notes in Computer Science*, Springer, pp. 764–783.

[158] Reparaz, O., De Meyer, L., Bilgin, B., Arribas, V., Nikova, S., Nikov, V., and Smart, N. P. CAPA: the spirit of beaver against physical attacks. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I* (2018), pp. 121–151.

[159] Rivain, M., and Prouff, E. Provably secure higher-order masking of AES. In *CHES* (2010), vol. 6225 of *Lecture Notes in Computer Science*, Springer, pp. 413–427.

[160] Rivest, R. L., Shamir, A., and Adleman, L. M. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21*, 2 (1978), 120–126.

[161] Roche, T., Lomné, V., and Khalfallah, K. *Combined Fault and Side-Channel Attack on Protected Implementations of AES*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 65–83.

[162] Roche, T., and Prouff, E. Higher-order glitch free implementation of the AES using secure multi-party computation protocols - extended version. *J. Cryptographic Engineering 2*, 2 (2012), 111–127.

[163] Romine, C. H. SHA-3 standard: Permutation-based hash and extendable output functions.

[164] Saha, S., Mukhopadhyay, D., and Dasgupta, P. Expfault: An automated framework for exploitable fault characterization in block ciphers. *IACR Trans. Cryptogr. Hardw. Embed. Syst. 2018*, 2 (2018), 242–276.

[165] Samwel, N., and Daemen, J. DPA on hardware implementations of Ascon and Keyak. In *Proceedings of the Computing Frontiers Conference* (New York, NY, USA, 2017), CF'17, ACM, pp. 415–424.

[166] San, I., and At, N. Compact Keccak hardware architecture for data integrity and authentication on FPGAs. *Information Security Journal: A Global Perspective 21*, 5 (2012), 231–242.

[167] Schaffenrath, D. Dom protected hardware implementations of keccak. `https://github.com/hgrosz/keccak_dom`, 2017.

[168] Schneider, T., Moradi, A., and Güneysu, T. ParTI - towards combined hardware countermeasures against side-channel and fault-injection attacks. In *CRYPTO (2)* (2016), vol. 9815 of *Lecture Notes in Computer Science*, Springer, pp. 302–332.

[169] Schwarz, M., Lipp, M., Moghimi, D., Bulck, J. V., Stecklina, J., Prescher, T., and Gruss, D. Zombieload: Cross-privilege-boundary data sampling. In *CCS* (2019), ACM, pp. 753–768.

[170] Seker, O., Eisenbarth, T., and Steinwandt, R. Extending glitch-free multiparty protocols to resist fault injection attacks. *IACR Cryptology ePrint Archive 2017* (2017), 269.

[171] Seker, O., Fernandez-Rubio, A., Eisenbarth, T., and Steinwandt, R. Extending glitch-free multiparty protocols to resist fault injection attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst. 2018*, 3 (2018), 394–430.

[172] Shamir, A. How to share a secret. *Commun. ACM 22*, 11 (1979), 612–613.

[173] Sijacic, D., Balasch, J., Yang, B., Ghosh, S., and Verbauwhede, I. Towards efficient and automated side channel evaluations at design time. In *PROOFS 2018, 7th International Workshop on Security Proofs for Embedded Systems, colocated with CHES 2018, Amsterdam, The Netherlands, September 13, 2018* (2018), pp. 16–31.

[174] Sundal, M., and Chaves, R. Efficient FPGA implementation of the SHA-3 hash function. In *2017 IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2017, Bochum, Germany, July 3-5, 2017* (2017), IEEE Computer Society, pp. 86–91.

[175] Tiri, K., and Verbauwhede, I. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *DATE* (2004), IEEE Computer Society, pp. 246–251.

[176] Touloupis, E., Flint, J. A., Chouliaras, V. A., and Ward, D. D. Study of the effects of seu-induced faults on a pipeline protected microprocessor. *IEEE Trans. Computers 56*, 12 (2007), 1585–1596.

[177] Trichina, E. Combinational Logic Design for AES SubByte Transformation on Masked Data. Cryptology ePrint Archive, Report 2003/236, 2003.

[178] van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., and Giuffrida, C. RIDL: rogue in-flight data load. In *IEEE Symposium on Security and Privacy* (2019), IEEE, pp. 88–105.

[179] Šijačić, D. Computer aided side channel analysis design environment (cascade).

[180] Wegener, F., and Moradi, A. A first-order SCA resistant AES without fresh randomness. In *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings* (2018), pp. 245–262.

[181] Winderickx, J., Daemen, J., and Mentens, N. Exploring the use of shift register lookup tables for Keccak implementations on Xilinx FPGAs. In *26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016* (2016), P. Ienne, W. A. Najjar, J. Anderson, P. Brisk, and W. Stechele, Eds., IEEE, pp. 1–4.

[182] Wolf, C., and Glaser, J. Yosys - a free verilog synthesis suite. *Proceedings of Austrochip* (2013).

[183] Wouters, L., Arribas, V., Gierlichs, B., and Preneel, B. A critical review of the CHES paper: Methodology for efficient CNN architectures in profiling attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst. 2020*, 3 (2020).

[184] Xilinx. 7 Series FPGAs.

[185] Xilinx. Virtex-6 libraries guide for hdl designs.

[186] Yen, S., and Joye, M. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Trans. Computers 49*, 9 (2000), 967–970.

[187] Yen, S., Kim, S., Lim, S., and Moon, S. A countermeasure against one physical cryptanalysis may benefit another attack. In *Information Security and Cryptology - ICISC 2001, 4th International Conference Seoul, Korea, December 6-7, 2001, Proceedings* (2001), pp. 414–427.

[188] Zhang, F., Guo, S., Zhao, X., Wang, T., Yang, J., Standaert, F., and Gu, D. A framework for the analysis and evaluation of algebraic fault attacks on lightweight block ciphers. *IEEE Trans. Information Forensics and Security 11*, 5 (2016), 1039–1054.

List of Publications

1. Arribas, V., Bilgin, B., Petrides, G., Nikova, S., and Rijmen, V. Rhythmic Keccak: SCA security and low latency in hw. *IACR Transactions on Cryptographic Hardware and Embedded Systems 2018*, 1 (2018), 269–290

2. Arribas, V., Nikova, S., and Rijmen, V. Guards in action: First-order SCA secure implementations of ketje without additional randomness. In *21st Euromicro Conference on Digital System Design, DSD 2018, Prague, Czech Republic, August 29-31, 2018* (2018), pp. 492–499

3. Arribas, V., Cnudde, T. D., and Sijacic, D. Glitch-resistant masking schemes as countermeasure against fault sensitivity analysis. In *FDTC* (2018), IEEE Computer Society, pp. 27–34

4. Reparaz, O., De Meyer, L., Bilgin, B., Arribas, V., Nikova, S., Nikov, V., and Smart, N. P. CAPA: the spirit of beaver against physical attacks. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I* (2018), pp. 121–151

5. Arribas, V., Nikova, S., and Rijmen, V. VerMI: Verification Tool for Masked Implementations. In *25th IEEE International Conference on Electronics, Circuits, and Systems* (Bordeaux,FR, 2018), IEEE, p. 4

6. De Meyer, L., Arribas, V., Nikova, S., Nikov, V., and Rijmen, V. M&m: Masks and macs against physical attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst. 2019*, 1 (2019), 25–50

7. Purnal, A., Arribas, V., and DeMeyer, L. Trade-offs in protecting Keccak against combined side-channel and fault attacks. In *Constructive Side-Channel Analysis and Secure Design - 10th International Workshop, COSADE 2019* (2019)

8. Arribas, V., Nikova, S., and Rijmen, V. Guards in action: First-order SCA secure implementations of KETJE without additional randomness. *Microprocessors and Microsystems - Embedded Hardware Design 71* (2019)

9. Arribas, V. Beyond the limits: SHA-3 in just 49 slices. In *FPL* (2019), IEEE, pp. 239–245

10. Arribas, V., Wegener, F., Moradi, A., and Nikova, S. Cryptographic fault diagnosis using verfi. *IACR Cryptology ePrint Archive 2019* (2019), 1312

11. Wouters, L., Arribas, V., Gierlichs, B., and Preneel, B. A critical review of the CHES paper: Methodology for efficient CNN architectures in profiling attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst. 2020*, 3 (2020)



FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF ELECTRICAL ENGINEERING
COSIC
Kasteelpark Arenberg 10 - bus 2452
B-3001 Leuven
victor.arribas@esat.kuleuven.be