CRYPTOGRAPHIC HARDWARE AGILITY FOR PHYSICAL PROTECTION

Dissertation

zur Erlangung des Grades eines Doktor-Ingenieurs
der Fakultät für Elektrotechnik und Informationstechnik
an der Ruhr-Universität Bochum

by Pascal Sasdrich
Bochum, October 2017
To my beloved wife Alina
and to my parents
for their love and endless support.
Abstract

Mass production and ubiquitous application of integrated logic circuits paved the way for the Digital Revolution and marked the beginning of the Information Age. Novel technologies and information processing devices enabled a plethora of new applications and innovations - ranging from Internet of Things to intelligent transportation systems - that dramatically change our habits and pervade our daily lives. With increasing gain and flow of information, security and privacy is ever becoming more important in particular for nearly ubiquitous small Microcontrollers and embedded devices. Since modern security mechanisms include a lot of complex cryptographic operations that encumber these constrained devices, efficient cryptography on hardware-based accelerators, including Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs), can help to ease this burden. Due to an unhindered and constant physical access and the static behavior and structure of these devices, side-channel attacks are a severe threat to the cryptographic implementations. Consequently, hardware agility, i.e., dynamic structures and constant changes in the circuit and its behavior, will render the static methods of Side-Channel Analysis irrelevant.

The first research contribution of this work is the development of novel low-area and high-performance cryptographic implementations for symmetric and asymmetric cryptosystems on reconfigurable devices. More precisely, this thesis investigates the application of particular features and aspects of modern FPGAs for high-efficient hardware-based cryptography. To this end, this work presents novel applications for Distributed Memory which re dedicate configuration memory of FPGAs as user-accessible storage elements in order to design a lightweight AES encryption core. Besides, this thesis at hand demonstrates the application of Block-RAMs and Digital Signal Processors for high-performance implementations of high-security Elliptic Curve Cryptography considered for the next generation of the TLS protocol. To counteract the threat of physical attacks, all presented designs include state-of-the-art side-channel countermeasures which are evaluated under the gray-box adversary model using a practical evaluation setup.

The second contribution of this thesis covers the topic of cryptographic hardware agility for physical protection. In particular, FPGAs as hardware platform for cryptographic accelerations provide the unique feature of run time reconfigurability to load and replace digital circuits on demand. In combination with algorithmic equivalences, the novelty of this work is the application of structural and algorithmic reconfiguration of cryptographic implementations in order to counteract passive Side-Channel Analysis. Consequently, this thesis investigates and evaluates different components of FPGAs that enable the restructuring of hardware circuits. On the path to a generic methodology for circuit reorganization and randomization, the thesis at hand analyzes algorithmic structures for equivalent circuit representations and demonstrates their application for amplification of Threshold Implementations. Eventually, the goal of a generic circuit randomization methodology for viable hardware agility is demonstrated employing the underlying concept and notion of White-Box Cryptography.
Keywords.

Cryptography, High-Efficient Implementations, Reconfigurable Hardware, FPGA, Side-Channel Analysis, Masking, Hiding, Dynamic Hardware Agility, Threshold Implementation, t-Test, White-Box Cryptography
Kurzfassung


Der zweite Kernaspekt dieser Arbeit behandelt das Thema der agilen Hardwarerekonfiguration für kryptographische Implementierungen zum Schutz gegen physikalische Angriffe. Insbesondere FPGAs bieten als Hardwareplattform die einzigartige Fähigkeit der Rekonfigurierbarkeit, um zur Laufzeit bei Bedarf beliebige digitale Schaltungen zu laden und zu ersetzen. Zusammen mit algorithmischen Äquivalenzen besteht das Novum dieser Arbeit in der Kombination von struktureller und algorithmischer Rekonfiguration für kryptographische Implementierungen zum Schutz gegen passive Seitenkanalanalyse. Demzufolge untersucht und evaluiert diese Dissertation verschiedene Komponenten der FPGAs, die eine Restrukturierung der Hardware-

**Schlagworte.**

Kryptographie, Hocheffizienzimplementierungen, Rekonfigurierbare Hardware, FPGA, Seitenkanalanalyse, Maskierung, Verschleierung, Dynamische Hardwareagilität, Threshold Implementierungen, t-Test, White-Box Kryptographie
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imprint</td>
<td>v</td>
</tr>
<tr>
<td>Preface</td>
<td>vi</td>
</tr>
<tr>
<td>Abstract</td>
<td>ix</td>
</tr>
<tr>
<td>Kurzfassung</td>
<td>xi</td>
</tr>
</tbody>
</table>

## 1 Preliminaries

### 1 Introduction

1.1 Motivation

1.2 Structure of this Thesis

1.3 Summary of Research Contributions

1.3.1 Efficient Implementations and Architectures

1.3.2 Hardware Agility for Physical Protection

## 2 Reconfigurable Hardware and Architectures

2.1 Field-Programmable Gate Arrays

2.1.1 General Purpose Components

2.1.2 Special Purpose Components

2.2 Optimization Metrics for Implementations

2.3 Architecture Strategies

## 3 Cryptography and Cryptographic Primitives

3.1 Symmetric Cryptography

3.1.1 Stream Ciphers

3.1.2 Block Ciphers

3.2 Asymmetric Cryptography

3.2.1 Integer Factorization Problem

3.2.2 Discrete Logarithm Problem

3.2.3 Discrete Logarithm Problem over Elliptic Curves

## 4 Physical Attacks and Countermeasures

4.1 Notations

4.2 Adversary Models

4.2.1 Black-Box Adversary

4.2.2 Gray-Box Adversary

4.2.3 White-Box Adversary
## Table of Contents

4.3 Side-Channel Analysis ........................................ 27  
4.3.1 Physical Side Channels ..................................... 27  
4.4 Evaluation Methods ........................................... 28  
4.4.1 Direct Observation ........................................ 29  
4.4.2 Differential Analysis ...................................... 29  
4.4.3 Statistical Leakage Assessment .............................. 30  
4.5 Countermeasures ............................................... 33  
4.5.1 Hiding .................................................. 33  
4.5.2 Masking ................................................ 35  

II Efficient Implementations of Cryptographic Primitives ...... 37  

5 Side-Channel Protected Low-Area AES Implementation ...... 39  
5.1 Introduction .................................................. 39  
5.1.1 Related work ............................................. 40  
5.1.2 Contribution .............................................. 41  
5.2 Design Considerations ........................................ 41  
5.2.1 Protection against Physical Attacks ....................... 42  
5.2.2 Pseudo Random Number Generation ....................... 42  
5.2.3 Modes of Operation and Decryption ....................... 42  
5.3 Implementation ............................................... 43  
5.3.1 Basic Encryption Core .................................... 43  
5.3.2 Adding Side-Channel Protection ........................... 46  
5.3.3 Pseudo Random Number Generation ....................... 47  
5.4 Results ..................................................... 48  
5.4.1 Comparison ............................................... 48  
5.5 Side-Channel Analysis ........................................ 49  
5.5.1 Evaluation under Lab Conditions ........................... 49  
5.5.2 Evaluation for Embedded Applications ..................... 51  
5.6 Conclusion .................................................. 51  

6 Exploring RFC 7748: Curve25519 with Side-Channel Protection ........................................ 53  
6.1 Introduction .................................................. 53  
6.1.1 Related Work ............................................. 54  
6.1.2 Contribution .............................................. 54  
6.2 Specification of CURVE25519 ................................ 55  
6.2.1 Field Arithmetic .......................................... 55  
6.2.2 Group Arithmetic ......................................... 55  
6.2.3 Point Multiplication ...................................... 56  
6.3 Design Considerations ........................................ 56  
6.4 Implementation ............................................... 58  
6.4.1 Single-Core Architecture ................................ 58  
6.4.2 Multi-Core Architecture .................................. 60
# Table of Contents

6.4.3 Side-Channel Countermeasures ............................................ 62
6.5 Results ................................................................................. 64
   6.5.1 Comparison .................................................................... 64
   6.5.2 Comparison to Other Work ............................................. 66
6.6 Evaluation ............................................................................ 67
   6.6.1 Detection of Scalar-Dependent Leakage ......................... 67
   6.6.2 Detection of Base-Point-Dependent Leakage .................... 69
6.7 Conclusions ........................................................................ 71

7 Exploring RFC 7748: Curve448 with Side-Channel Protection 73
   7.1 Introduction ........................................................................ 73
      7.1.1 Related work ............................................................. 74
      7.1.2 Contribution ............................................................... 74
   7.2 Specification of Curve448 .................................................. 74
      7.2.1 Field Arithmetic ......................................................... 75
      7.2.2 Group Arithmetic ....................................................... 75
      7.2.3 Point Multiplication ................................................... 75
      7.2.4 Side-Channel Protection ............................................. 76
   7.3 Design Rationales ............................................................... 76
      7.3.1 Implementation Objectives ......................................... 77
      7.3.2 General Optimization Strategies ................................. 77
      7.3.3 Field Arithmetic Optimizations .................................. 78
   7.4 Implementation Details ...................................................... 79
      7.4.1 Field Arithmetic Unit ................................................. 80
      7.4.2 Side-Channel Countermeasures ................................... 81
   7.5 Results .............................................................................. 81
      7.5.1 Implementation Results ............................................. 81
      7.5.2 Comparison ............................................................... 82
   7.6 Evaluation ........................................................................ 83
      7.6.1 Detection of Scalar-Dependent Leakage ..................... 83
      7.6.2 Detection of Base-Point-Dependent Leakage ............... 85
   7.7 Conclusion ........................................................................ 87

III Hardware Agility using Dynamic Reconfiguration 89

8 Dynamic Logic Reconfiguration .............................................. 91
   8.1 Introduction ...................................................................... 91
      8.1.1 Contribution .............................................................. 92
   8.2 Preliminaries .................................................................... 92
      8.2.1 Configurable Look-Up Table ....................................... 92
      8.2.2 Reconfigurable Function Table ................................... 93
      8.2.3 Basic Architecture ..................................................... 93
Table of Contents

8.3 Countermeasures ........................................... 94
  8.3.1 Random S-box Decomposition ......................... 94
  8.3.2 Boolean Masking ..................................... 95
  8.3.3 Register Precharge .................................. 96
8.4 Practical Evaluation ...................................... 96
  8.4.1 No Countermeasure .................................. 97
  8.4.2 Single Countermeasure ................................. 97
  8.4.3 Combination of Countermeasures ....................... 98
8.5 Conclusion .................................................. 100

9 Pitfalls of Memory Primitives 103
  9.1 Introduction ............................................ 103
    9.1.1 Contribution ....................................... 104
  9.2 Preliminaries ........................................... 104
    9.2.1 Memory Primitives .................................. 104
    9.2.2 Randomized Look-Up Tables ......................... 106
    9.2.3 Block Memory Content Scrambling .................... 107
  9.3 Design .................................................... 108
    9.3.1 Masking Architecture ................................ 108
    9.3.2 Comparison of S-box designs ....................... 109
  9.4 Evaluation ................................................ 110
    9.4.1 Results ............................................. 110
  9.5 Conclusion ................................................ 112

10 Affine Equivalence and its Applications 113
  10.1 Introduction ............................................ 113
    10.1.1 Contribution ....................................... 114
  10.2 Background ............................................... 114
    10.2.1 S-Box Decomposition ................................ 114
    10.2.2 Affine Equivalence ................................ 116
  10.3 Design Considerations .................................. 116
    10.3.1 Threshold Implementation of PRESENT ............... 116
    10.3.2 Searching for the Affine Triples .................... 118
    10.3.3 Selecting Precomputed Affine Triples ............... 118
    10.3.4 Generating Affine Triples On-the-fly ............... 119
    10.3.5 Comparison of the Approaches ....................... 120
  10.4 Evaluation ................................................ 121
    10.4.1 Results ............................................. 122
  10.5 Discussions ............................................. 124
  10.6 Conclusion ................................................ 126

11 White-Box Cryptography in the Gray Box 127
  11.1 Introduction ............................................ 127
    11.1.1 Contribution ....................................... 128
    11.1.2 Related Work ....................................... 128
# Table of Contents

11.2 Background .................................................. 129
   11.2.1 White-Box Cryptography .................................. 129
   11.2.2 White-Box Implementation of AES ...................... 129
11.3 FPGA Implementation ........................................ 133
   11.3.1 White-Box Architecture in Hardware .................. 134
   11.3.2 Performance Evaluation ................................. 135
11.4 Side-Channel Analysis ........................................ 135
   11.4.1 Differential Computational Analysis Attack .......... 135
   11.4.2 Differential Power Analysis Attack ................... 137
   11.4.3 Mathematical Foundations .............................. 138
   11.4.4 How to Avoid such Attacks ............................. 140
11.5 Conclusion .................................................. 141

12 Randomizing Cryptographic Implementations .................. 143
   12.1 Introduction ............................................... 143
   12.1.1 Contribution ............................................. 144
   12.1.2 Previous Work ............................................ 144
12.2 Background .................................................. 145
   12.2.1 Directed Graphs .......................................... 145
12.3 Methodology ................................................ 145
   12.3.1 Generic Approach ........................................ 146
   12.3.2 Morphing Algorithm for Cryptographic Implementations . 146
   12.3.3 Applicable encoding functions .......................... 146
   12.3.4 Verification and semantic equivalence checking ........ 147
12.4 Case Study: PRESENT Threshold Implementation ............ 148
   12.4.1 Adversary Model ........................................ 148
   12.4.2 Design Considerations ................................... 148
   12.4.3 Practical Implementation on Reconfigurable Hardware . 151
   12.4.4 Comparison .............................................. 152
12.5 Practical Side-Channel Evaluation ........................... 152
   12.5.1 Results .................................................. 153
12.6 Conclusion .................................................. 155

IV Conclusion ..................................................... 157

13 Conclusion and Future Work .................................. 159
   13.1 Conclusion ................................................ 159
   13.2 Directions for Future Work ................................ 160
      13.2.1 Efficient Implementations ............................. 160
      13.2.2 Hardware Agility and Physical Protection ............ 160

V Appendix ......................................................... 163
# Table of Contents

14 Component functions of the PRESENT S-box  
14.1 Uniform TI of $Q_{12}$ using 3 shares .................................................. 165  
14.2 Uniform TI of $Q_{204}$ using 3 shares .................................................. 165  
14.3 Uniform TI of $Q_{299}$ using 3 shares .................................................. 166

Bibliography  
List of Abbreviations  
List of Figures  
List of Tables  
About the Author  
Publication and Academic Activities
Part I

Preliminaries
Chapter 1
Introduction

This chapter provides a brief introduction to the topic of efficient cryptographic implementations on re-programmable hardware devices. On that basis we review the threat model of physical attacks for static cryptographic hardware implementations and motivate the idea and concept of dynamic hardware agility as suitable countermeasure. Both aspects, efficient cryptographic implementations and dynamic countermeasures against physical attacks, form the basis of this thesis and our research contribution.

Contents of this Chapter

1.1 Motivation ....................................................... 3
1.2 Structure of this Thesis ........................................ 5
1.3 Summary of Research Contributions ......................... 6

1.1 Motivation

The Digital Revolution and the onset of the Information Age has initiated a rapid transformation of our society towards a modern information society. The creation, distribution, integration, application, and manipulation of information has become a fundamental part of our daily life that increasingly pervades and determines economy, education, and governmental processes. An ever closer integration of the physical world into computer-based systems and the constant creation and flow of information demand for an extensive and distributed infrastructure of smart devices. These Cyber Physical Systems (CPSs) are the backbone of our modern information society that open up entirely new opportunities and applications such as the Internet of Things (IoT), intelligent transportation systems, smart homes, cities, and grids, and industrial automation.

Nowadays, many industrial and commodity products integrate smart systems and devices to provide additional functionalities in order to collect, process, transfer, and store information. In combination with a virtually unlimited interconnection via the Internet (far beyond simple machine-to-machine communication), these embedded devices can improve the efficiency and accuracy of the products while increasing the economic benefit and reducing human interactions. Hence, the ushering of automation in all parts of our society provides many opportunities to facilitate and enrich our daily life, however, it still brings many challenges that have to be solved.
Through the continuous reduction of semiconductor structural sizes and the accompanying compression and integration of electronic circuits, digital control of consumer products using embedded systems has become increasingly economical. Many smart devices incorporate Microcontrollers (µCs) or a System-on-a-Chip (SoC) particularly designed for embedded applications. These small, pervasive computing devices are captivating due to their generality, versatility, and low per-unit costs. However, for very constrained scenarios, Application-Specific Integrated Circuits (ASICs) are a considerably more suitable and powerful alternative, despite higher non-recurring costs, providing an integrated circuit architecture heavily optimized for a specific problem rather than for general purpose applications. Unfortunately, this application specific optimization is accompanied by a predetermined and static circuit structure which takes the user any possibility to replace or change the function or application. In this regard, Field-Programmable Gate Arrays (FPGAs) are reconfigurable integrated circuits which attempt to close the gap between µCs and ASICs. Based on a regular array of programmable logic blocks and interconnections, FPGAs dynamically allow mapping and emulating of user-defined hardware circuits for various applications. Due to these advantages, FPGAs have become increasingly popular as embedded devices in recent years.

As our modern society is changing and the volume of ubiquitous data is growing continuously, information security is becoming a crucial part of embedded applications. In order to prevent unauthorized access, disclosure, modification, or exploitation of sensitive information, appropriate measures to ensure confidentiality, authenticity, integrity, and privacy of information have to be taken. As a consequence, cryptography is used in many applications, devices, and protocols since it provides these properties that can solve most of the problems. But, although costs for embedded devices are constantly decreasing, security still is considered as an expensive and dispensable extra service. Despite bad precedents, cryptographic functions are often excluded from applications for efficiency and economic reasons. Hence, efficient cryptographic implementations particularly designed for embedded devices and systems are an important objective.

With the onset of pervasive, ubiquitous computing by means of embedded devices and systems, the threat situation of cryptography particularly in embedded applications has dramatically changed. Well-known and standardized cryptographic algorithms, which have been thoroughly examined and evaluated for theoretical security against cryptanalysis, are again vulnerable to practical attacks. Due to physical access to cryptographic operations running on embedded devices, an adversary can simultaneously and passively observe the device behavior and its physical characteristics. Granted with these additional insights on the implementations and operation environment, any cryptographic algorithm no longer behaves as a black box but as a gray box that provides additional side-channel information.

Endowed with such powerful abilities, gray-box adversaries are an immense and severe threat for implementations of (theoretically sound) cryptographic algorithms and appropriate countermeasures are mandatory for any embedded device or system in the field. To this end, a plethora of different types of countermeasures against Side-Channel Analysis (SCA) have been proposed and implemented. The essential task of these countermeasures is the prevention of data-specific behavior in order to avoid leakage of side-channel information. In practice, two different approaches, hiding and masking, have prevailed. While the first schemes focus on the reduction of the Signal-to-Noise Ratio (SNR) of observable side-channel information, the second type of
schemes randomizes sensitive intermediate data in order to break any dependencies between observation and sensitive data.

Although cryptography is deployed to ensure confidentiality of sensitive information and data, it is evident that the physical circuit of the embedded system is not protected and publicly accessible. Consequently, physical access to an embedded system allows a technically and temporally unconstrained adversary to identify, classify, inspect and disassemble any security features of the system. Due to the static structure and behavior of the circuit, the adversary can orchestrate a temporally decoupled and combined structural examination spread over many devices in order to circumvent the security features.

By implication, a dynamic structure and behavior of the underlying hardware can impede analysis of the security features such that any attempt to apply static attack principles will be in vain. Particularly, restructuring and reorganization of circuits and embedded systems helps to move cryptographic implementations in space and time in order relocate and delay operations on sensitive information. Hence, in combination with existing side-channel countermeasures (i.e., masking and hiding), cryptographic hardware agility prevents the characterization of internal processes and creation of suitable attack models. More precisely, along with the particular characteristics of modern FPGAs to support full or partial reconfiguration during run time, this provides an optimal basis to design and implement self-modifying but functionally invariant cryptographic hardware circuits for physical protection.

1.2 Structure of this Thesis

The main body of this thesis is organized in two thematic parts in order to discuss the main topics of this work independently. While the first part describes efficient implementations and architectures for both symmetric and asymmetric cryptographic primitives, the second part discusses results on hardware agility for physical protection using structural and algorithmic reconfiguration. In detail, the thesis is structured as follows:

Part I: Preliminaries. This part is dedicated to a brief introduction of the research motivation and contributions of this thesis regarding efficient implementations of cryptographic primitives on reconfigurable devices and their protection against physical attacks. Accordingly, in Chapter 2, we provide general background information on modern reconfigurable devices and highlight their most important features. Then, in Chapter 4, we introduce the notion of SCA, its adversary model and essential evaluation methods before describing the concepts of common side-channel countermeasures.

Part II: Efficient Implementations of Cryptographic Primitives. In Chapter 5 we describe a low-area AES-128 encryption core that is implemented on reconfigurable hardware and immensely benefits from Distributed Memory. Based upon this basic architecture, we propose a side-channel protected, self-contained encryption core that maintains its lightweight characteristics but provides physical protection along with random number generation abilities.

The second half of this part deals with high-performance public-key implementations of the RFC 7748 [LHT16] on reconfigurable devices. More precisely, in Chapter 6 we present the
Chapter 1 Introduction

first implementation of the elliptic curve Curve25519 on reconfigurable hardware. Moreover, we discuss the integration of protection mechanisms against physical attacks and a multi-core architecture for high-performance applications. Consequently, in Chapter 7 we present the hardware implementation of the second elliptic curve Curve448 that was proposed in RFC 7748 along with its protection against physical attacks.

Part III: Hardware Agility using Dynamic Reconfiguration. At first, this part highlights the opportunities of Distributed Memory features on modern FPGAs for side-channel security as well as potential pitfalls that may occur during the design phase and can affect the physical protection of the device. To this end, we analyze and evaluate structurally reconfigurable hardware architectures of PRESENT (Chapter 8) and AES (Chapter 9) for their physical security potential.

Furthermore, this part covers different approaches and countermeasures against higher-order side-channel analysis that combine first-order secure Threshold Implementations (TIs) with algorithmic reconfiguration. In particular, in Chapter 10 we discuss the concept of affine equivalences of Boolean functions and its application to provide higher-order protection based on continuous restructuring of the nonlinear part, i.e., the S-box, of a PRESENT encryption core. However, in order to find a generic methodology that is not limited to particular structures, we review the concept of White-Box Cryptography in Chapter 11. Though we can demonstrate that the notion of White-Box Cryptography likewise suffers from its static behavior and is vulnerable to gray-box attacks, still we can adapt and extract ideas in order to construct a generic methodology for dynamic hardware modifications in Chapter 12.

Part IV: Conclusion. To conclude this thesis, we review the presented results and discuss starting points and novel directions for future work in Chapter 13.

1.3 Summary of Research Contributions

The research contributions in this thesis are related to efficient cryptographic hardware architectures on reconfigurable devices and their protection against physical attacks based on hardware agility. Most of the results are part of conference or journal publications [SG14, SG15, SMMG15a, SMMG15c, SMG15a, SMG16b, SG16a, SMG17, SG17]. Note, that all of these publications cover an aspect of the main topics and are the basis for chapters or sections of this thesis, but sometimes the content is rearranged for better readability. Further projects and publications in which the author participated, but that go beyond the scope of this thesis, are excluded from this thesis but can be found in [GRSW14, BJK+16a, JMPS17a]. In the following, we briefly summarize and highlight our contribution for each chapter.

1.3.1 Efficient Implementations and Architectures

Over the past decades, cryptography has become a prosperous science that yielded many novel techniques and concepts for securing electronic information and protecting personal digital assets. Despite its success story, it is still often considered dispensable for industrial products since it involves additional expenses and its importance is not directly tangible. Modern cryptographic systems, usually divided into public-key and private-key algorithms, often involve
1.3 Summary of Research Contributions

complex mathematical operations and require a lot of additional area on embedded chips and systems. Since inefficient cryptography is often excluded from final products, even despite bad experiences, it is important to provide high-efficient cryptographic designs and implementations (particularly in terms of area and performance). Accordingly, this thesis presents different contributions for low-area and high-performance implementations of symmetric and asymmetric cryptosystems on reconfigurable devices.

Side-Channel Protected Low-Area AES [SG16a]

Since its standardization in 2001, AES has become the predominant private-key block cipher worldwide which is recommended and used for many cryptographic applications. Consequently, it has been analyzed and studied in-depth leading to a wealth of optimized, high-efficient software and hardware implementations. Despite this fact, we introduce three novel, ultra-lightweight AES-128 implementation specifically tailored for modern reconfigurable hardware. The basic proposal presents the smallest key-agile AES-128 encryption core reported in open literature that uses just 21 slices of a Spartan-6 FPGA (without any additional memory) while still providing a moderate throughput of 9.12 Mbps. Since the basic design heavily exploits the Distributed Memory features of the Spartan-6 FPGA, we can easily show that this architecture almost inherently supports shuffling as side-channel countermeasure at minimal extra costs of three additional slices and a reduced throughput of 7.82 Mbps. Our third proposal combines the side-channel protected core with random number generation abilities in order to construct a protected, self-contained, but lightweight AES-128 encryption core within 28 slices that still provides a throughput of 4.35 Mbps. Eventually, we evaluate and proof the effectiveness and security of our architectures under SCA using Correlation Power Analysis (CPA) while still maintaining a lightweight setting for all proposed designs.

Side-Channel Protected Curve25519 [SG14, SG15] and Curve448 [SG17]

Accordingly, Elliptic Curve Cryptography (ECC) has become the predominant cryptographic system for public-key security-critical applications, such as efficient key agreement and digital signature schemes. However, as indicated before, ECC involves complex mathematical operations and modular arithmetic that is a particular burden for small and constrained devices. In this context, Request for Comments (RFC) 7748 [LHT16] proposes two elliptic curves over prime fields – Curve25519 and Curve448 – which are in particular considered for Transport Layer Security (TLS) in the context of IoT and offer a high level of practical security. Despite their efficient and slim design chosen for highly efficient software implementations, neither of the two was particularly constructed for efficient hardware implementations. Moreover, physical protection against side-channel attacks was considered thoroughly during the design phase. To this end, we give evidence that both elliptic curves can indeed be efficiently and securely implemented in reconfigurable hardware.

Starting with Curve25519, we show that it is likewise competitive on FPGAs even when countermeasures to thwart SCAs are included. Our basic single-core DSP-based architecture achieves a maximal performance of more than 2,500 variable base point scalar multiplications per second on a Xilinx XC7Z020 FPGA, and still more than 2,400 operations per second with a moderate increase in logic resources and in presence of state-of-the-art side-channel countermeasures. Extending our basic design to a multi-core architecture we can boost the maximum
performance to more than 32,000 point multiplications per second using our unprotected design and still more than 27,500 operations per second including a mix of side-channel countermeasures to impede Simple Power Analysis (SPA) and Differential Power Analysis (DPA).

Further on, we give evidence that even the second candidate Curve448, particularly designed for high-security applications beyond the security level of 128 bits, still maps efficiently to reconfigurable hardware. In its basic configuration, our novel architecture for Curve448 can compute more than 1,000 point multiplications per second with only 1,580 logic slices and 33 DSP units of a Xilinx XC7Z020 FPGA. Including side-channel protection mechanisms, the resource occupation increases to 1,648 logic slices and 35 Digital Signal Processor (DSP) units while the performance decreases to 708 operations per second. Eventually, we evaluate both architectures including side-channel countermeasures for their rigidity against physical attacks using a state-of-the-art leakage assessment methodology.

1.3.2 Hardware Agility for Physical Protection

Granted with unlimited physical access to embedded devices hosting cryptographic routines, gray-box adversaries are a severe threat for embedded and real-world applications. The static behavior and structure of the underlying circuitry facilitates an in-depth analysis of the cryptographic implementation, eventually leading to a suitable model for the physical characteristics of the circuit and implementation. Adapting and extending the notion of self-modifying binaries in the world of software in order to prevent static analysis and disassembly, the concept of self-modifying circuits aims to prevent classification and modeling of the physical characteristics. In addition, temporal and spatial mismatch of observations during run time rule out the application of static methods for SCA. In the course of this thesis, we discuss different approaches to realize the concept of dynamic hardware modifications in order to achieve cryptographic hardware agility on reconfigurable devices. Starting with applications of structural features of modern FPGAs, we then discuss the utilization of algorithmic structures of modern encryption schemes before we ultimately present a generic algorithmic approach based on the concept of White-Box Cryptography.

Hardware Structures for Dynamic Reconfiguration [SMMG15a, SMMG15c]

Modern FPGAs captivate with unique features for dynamic and partial run time reconfiguration in order to load hardware circuits just on demand or reload and replace portions. This implies, that completely different circuits might operate at the same location at different times or the same circuit might operate at different locations of the FPGA. This temporal and spatial uncertainty makes it difficult for an observer or attacker to predict operations and behavior of the device. In addition, modern features of Xilinx FPGAs including Distributed Memory and Configurable Look-Up Tables (CFGLUTs), easily support a nearly instantaneous exchange of hardware internals e.g., for cipher implementations. Hence, in this part we analyze the opportunities of Distributed Memory resources in order to build a novel hardware implementation of the lightweight cipher PRESENT with built-in side-channel countermeasures. In addition, we practically evaluate the rigidity of this dynamic logic reconfiguration against SCA. Our measurements based on a Spartan-6 platform provide evidence that even with 10 million power traces, the detection of first-order side-channel leakage is not possible.
1.3 Summary of Research Contributions

On this account we also review the Block Memory Content Scrambling (BMS) countermeasure, presented at CHES 2011, which enables an effective way of first-order side-channel protection for cryptographic primitives using run-time reconfiguration of randomized look-up tables. Considering three different instances of Distributed Memory (RAM32M, RAM64M, and RAM256X1S) and a Block-RAM (BRAM) primitive (RAMB8BWER), we provide a detailed study of alternatives to implement a dynamic first-order masking scheme for AES including a detailed comparison of different operation and reconfiguration times. Practical evaluations of the resistance against first-order SCA reveal deep problems and pitfalls of the Distributed Memory primitives and lead to the conclusion that solely the BRAM primitive can be used to realize a first-order side-channel protected implementation following the BMS concept.

Algorithmic Structures for Dynamic Reconfiguration [SMG15a]

Threshold Implementations solve the problem of glitches in masked hardware circuits, but only are able to avoid first-order side-channel leakage. Hence, motivated by the development of SCA countermeasures which can provide security only up to a certain order, defeating higher-order attacks has become amongst the most challenging issues. Consequently, the extension of TIs to Higher-Order Threshold Implementations (HO-TIs) aims at counteracting SCA attacks at higher statistical orders but might be limited to univariate scenarios. Although a $d$-order TI is vulnerable to an attack at order $d+1$, it is harder to mount an attack at increasing orders with respect to the number of traces as well as the sensitivity to noise.

To this end, we look at the feasibility of higher-order attacks on first-order TI from another perspective. Instead of increasing the order of resistance by employing HO-TIs, we go toward introducing structured randomness into the implementation. Our construction, which is a combination of masking and hiding, is dedicated to TI designs and deals with the concept of affine equivalences of Boolean functions, in particular for 4-bit nonlinear S-boxes. We show that the area overhead of our construction which constantly restructures the nonlinear layer of the PRESENT encryption is paid off by its ability to avoid higher-order leakages. In addition, such a combination hardens our design practically against higher-order attacks so that the leakage cannot be successfully exploited.

Randomizing Cryptographic Implementations [SMG16b, SMG17]

In a white-box scenario, an adversary has full control over the execution process and the entire environment. Implementations of White-Box Cryptography aim to protect a secret key. Its fundamental principle is the mapping of a cryptographic architecture, including the secret key, to a number of encoded tables that shall resist the inspection and decomposition of an attacker. In a gray-box scenario, however, the property of hiding required implementation details and the key from the attacker could be used as a promising mitigation strategy against SCA.

In this part, we present a first white-box implementation of AES on reconfigurable hardware for which we evaluate this approach assuming a gray-box attacker. We show that – unfortunately – such an implementation does not provide sufficient protection against SCA attackers due to its static configuration. We continue our evaluations by a thorough analysis of the observed leakage, and present additional results which can be used to build stronger white-box designs.

Eventually, we review the fundamental notions of White-Box Cryptography in order to combine them with a first-order secure TI of a symmetric cryptosystem again as an alternative
concept to HO-TI. Therefore, we investigate the idea of *dynamic hardware modifications*, i.e., random changes and transformations of cryptographic implementations regardless of algorithmic structures, in order to render higher-order attacks impractical. In a first step, we present a generic methodology which can be applied to (almost) every cryptographic implementation. In order to investigate the effectiveness of our proposed strategy, we use an instantiation of our methodology that adapts the idea on encoded look-up tables from White-Box Cryptography and applies this construction to a first-order secure TI. Further, we show that dynamically updating cryptographic implementations during operation provides the ability to avoid higher-order leakages to be practically exploitable.
Chapter 2
Reconfigurable Hardware and Architectures

In this chapter, we review the general concept and structure of FPGAs and provide detailed background information on general purpose and selected special purpose components of modern Xilinx FPGAs. In addition, we discuss general optimization strategies for hardware-based implementations and describe the most common design strategies for cryptographic hardware architectures.

Contents of this Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Field-Programmable Gate Arrays</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Optimization Metrics for Implementations</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Architecture Strategies</td>
<td>16</td>
</tr>
</tbody>
</table>

2.1 Field-Programmable Gate Arrays

Field-Programmable Gate Arrays are re-programmable highly regular integrated circuits and semiconductor devices which basically consist of an array of programmable logic blocks connected via programmable interconnections in a grid-like structure. Nowadays, most FPGAs are based on volatile Static Random Access Memory (SRAM) technology and have to be re-programmed after every power-up (unlike flash-based FPGAs) given a binary configuration (bitstream). Moreover, the grid-like structure of FPGAs has prevailed as common concept and devices from different vendors (e.g., Xilinx, Altera, Lattice, or Microsemi) mainly differ in the architecture and characteristics of low-level components, i.e., the programmable logic functions and routing network. Since, in the course of this work, we focus on Xilinx FPGAs, we restrict the description and discussion in order to highlight only particular features of recent devices.

2.1.1 General Purpose Components

FPGAs can realize almost any digital circuits of arbitrary complexity and are used in many areas that usually require flexibility and fast signal processing. For this purpose, FPGAs provide numerous generic logic components, i.e., synchronous storage elements and re-programmable function generators, that can implement any digital circuit and Boolean logic function.
Configurable Logic Block

Xilinx FPGAs aggregate all programmable logic functions in Configurable Logic Blocks (CLBs) that are arranged in a grid-like structure in order to implement sequential and combinatorial circuits. Each CLB incorporates two Logic Slices (LSs) and is connected to the global routing network via a Switch Matrix (SM). In turn, all LSs implement the basic function generators, but, depending on the device family and generation, different types of LS with additional features and characteristics are implemented to enhance the basic LS architecture.

Slice-X. Starting with Xilinx' 5-series FPGAs, the basic LS architecture (only available on Spartan devices) implements four different function generators for Boolean logic circuits in terms of Look-Up Tables (LUTs) and eight adjacent Flip-Flops (FFs) as storage elements.

Look-Up Table: Each LUT contains 64-bit configuration memory, 6 inputs and 2 independent outputs (O6 and O5) in order to implement either 6 : 1 Boolean functions or two 5 : 1 Boolean functions with shared inputs. Figure 2.1 shows the schematic layout of a LUT-6. Using the 64 configuration FFs and the multiplexer-tree below, the correct result of the Boolean function is generated based on the input bits.

Flip-Flop: Since each LUT has 2 independent outputs, each LS implements two storage elements per LUT in order to hold the results or for clock synchronization of the implemented circuit. In addition, all FFs attached to the O6-port of the LUTs can be configured either as edge-triggered D-FF or as level-sensitive latch while the remaining storage elements are only available as D-FFs and cannot be accessed if a latch is implemented.

Slice-L. Logic Slices with special abilities and characteristics in terms of logic handling are called Slice-L. These LSs provide the same basic features (4 LUTs, 8 FFs) as the standard Slice-X architecture but are enhanced for arithmetic operations and equipped with additional logic resources for carry processing and wide multiplexers in order to combine all LUTs to build either two 7 : 1 or a single 8 : 1 Boolean function as LUT.

Slice-M. The most powerful type of LSs is called Slice-M, augmenting the capabilities of the Slice-L version by distributed memory features called LUTRAM. Each instance of this LS can implement either 256-bit memory as synchronous Random Access Memory (RAM) (64 bit for each LUT per LS) or 128-bit shift registers (32 bit for each LUT) by rededicating the
configuration memory of the LUTs and relinquishing the option to implement Boolean functions. Xilinx provides several instances to implement such RAM components or shift registers, ranging from $256 \times 1$-bit to $64 \times 4$-bit LUTRAM or a single 128-bit to eight 16-bit shift registers per LS. In addition, depending on the chosen configuration, single port, dual port (simple or true), or quad port RAM is available. Figure 2.1 shows the schematic layout of a single 6-input LUT including the shift register mode. In particular, the additional ports $\text{DI}_0$ and $\text{DI}_1$ can be used to shift new data into the configuration memory and reuse the entire LUT as shift register.

### 2.1.2 Special Purpose Components

The major application field of FPGAs is high-performance digital signal processing. Although general purpose components of modern FPGAs provide high flexibility and good performance, additional modules have been included particularly to support fast processing of more complex and memory intensive operations.

**Block-RAM**

Modern Xilinx FPGAs are mainly designed and employed for rapid processing of big data volumes. Consequently, all devices provide additional on-chip memory in terms of Block-RAM (BRAM) to support efficient data handling. Over the last decade, with increasing complexity of recent FPGAs, BRAM modules have been improved immensely, now providing massive amounts of storage and a plethora of auxiliary features.

For instance, recent 7-series FPGAs (predominantly use in this work) are equipped with 36kb versatile, configurable BRAM units. Each BRAM instance can be configured as single 36kb general-purpose storage element (as shown in Figure 2.2) or split into two independent 18kb RAM or Read-Only Memory (ROM) modules. In addition, cascading of two neighboring BRAMs allows to create up to 64kb RAM instances. Moreover, with broad control over the access interface, each BRAM can be customized for virtually every application and purpose. In particular, each module be configured as RAM ranging from $32K \times 1$-bit to $1K \times 36$-bit in...
true dual-port mode (providing two independent, clock-synchronous access ports) or even 512 × 72-bit in simple dual-port mode. Along with enhanced features, such as optional pipeline registers, Error Correction abilities and synchronous, dual-clock First In - First Out (FIFO) options, modern BRAMs provide compact and low-power on-chip memory beneficial for many applications.

**Digital Signal Processing Units**

Due to data-intensive computing processes, digital signal processing is a major field of application which benefits from the capabilities of modern FPGAs and their provided resources. Hence, besides sufficient on-chip storage, FPGAs implement large numbers of full-custom, low-power DSP slices based on binary multiplication and accumulation units. Consequently, DSP modules are primarily dedicated for digital signal processing applications, however, support a variety of different configuration and modes in order to maintain versatility for various other applications.

Starting with the 7-series, each DSP (as shown in Figure 2.3) incorporates a 25-bit × 18-bit two’s complement binary multiplier enhanced by a 25-bit preaddition and a 48-bit ensuing accumulation stage. Optionally, the accumulator can be configured to implement a Arithmetic Logic Unit (ALU) which provides various different logic functions (e.g., XOR, XNOR, AND, NAND, OR, NOR) that can be selected dynamically during run time. Moreover, the accumulation stage can be configured as Single Instruction Multiple Data (SIMD) unit either providing dual 24-bit or quad 12-bit addition, subtraction or accumulation functionality.

Eventually, enhanced by a pattern detection circuit, various pipeline registers and cascading options, each DSP implements a small but versatile arithmetic processing unit primarily dedicated for digital signal processing applications. However, due to several configuration modes that even allow time multiplexed applications of the DSPs, each core can support a plethora of additional applications apart from digital signal processing.

**2.2 Optimization Metrics for Implementations**

In general, hardware designers will strive to optimize (cryptographic) architectures and designs to their greatest possible extent with respect to efficiency and security. Hence, disregarding the aspect of security, efficiency can be considered as one of the most important objectives...
for cryptographic implementations. However, since efficiency can be considered from different points of view, in this section we briefly discuss the most common metrics to assess the efficiency of (cryptographic) hardware designs.

**Resource Cost:** For FPGAs, the measurement of resource cost or area is somewhat difficult since the actual physical area of the chip is independent of the implemented hardware design. Hence, area evaluations of FPGA designs are usually based on the number of basic resources, such as LUTs, FFs, or LSs, which are instantiated and necessary to implement the design. Still, this metric is limited and does not take more advanced resources (e.g., BRAM, DSP) and optimization strategies into account. Further, comparison of devices from different families or vendors is also hardly possible.

**Latency:** Besides the physical area and number of resources, the latency until a computation result becomes available at the output of a hardware circuit is an important efficiency criterion for (cryptographic) implementations. In particular for synchronous designs (as well as most software-based implementations), the number of clock cycles to process input data is a common metric to compare the efficiency of different implementations and designs in terms of latency.

**Time:** Depending on the critical path, i.e., the longest delay between two synchronous elements within a circuit, the maximum frequency of different implementations will vary. Hence, either by dividing the amount of cycles (i.e., the latency) by the maximum operating frequency or by assuming a standardized operating frequency (e.g., 100 kHz), the amount of time for an operation can be calculated as a scaled metric.

**Throughput:** Given the time to perform a (cryptographic) operation, the throughput of a design is defined as the rate at which the output of an implementation is produced (usually expressed in bits per second). In other words, the number of output bits is multiplied by the operation frequency and divided by the latency in order to derive the final throughput as a combined metric.

**Efficiency:** Hardware efficiency is usually expressed either by the throughput to area ratio or as the time area product. In both cases, the area metric is combined with time in order to establish a joined metric for efficiency. However, due to the difficulty of defining an appropriate area metric for FPGAs, efficiency is rarely used, if at all, as throughput per slices.

Besides implementation dependent properties in terms of area, time and throughput, physical characteristics are sometimes considered as additional efficiency metrics. In particular, the power or energy consumption are often considered optimization metrics for hardware implementations. Although these characteristics are partially limited by the physical properties of the underlying technology and circuit, a significant portion still is affected by the implementation and hence can be influenced by hardware designers.

**Power:** Generally, the total power dissipation of an integrated Complementary Metal-Oxide-Semiconductor (CMOS) circuit depends on various factors, e.g., core supply voltage, operation frequency, switching activity, and technology dependent factors. Since hardware
designers using FPGAs cannot influence and control technology dependent factors but are limited to the manufacturer-determined properties of the devices, power optimization on FPGAs mainly focuses on the former factors. In particular, since the switching activities are directly proportional to active components and modules, decreasing the area simultaneously decreases the power consumption of the device. Therefore, reduction of the maximum operation frequency and area minimization are common approaches to optimize the power dissipation of a hardware circuit.

**Energy:** The energy consumption of a device is determined by the power dissipation over a certain time period. Hence, power minimization strategies can help to minimize the energy consumption although some applications may allow a high peak power consumption but in total require a low energy consumption.

### 2.3 Architecture Strategies

Different applications demand for different architecture strategies in order to tailor a given design for a particular optimization goal. Whereas an implementation for a low cost passive smart device (e.g., a Radio-Frequency Identification (RFID) tag) usually is optimized for small area and power consumption, communication intensive devices (e.g., a RFID reader processing many tags in parallel) will be optimized for high performance and throughput ignoring area and power consumption scores. Hence, with the wide range of different application scenarios, three major hardware architecture strategies can be applied: parallel (unrolled or pipelined), serial (with different levels of serialization), and hybrid approaches.

**Parallel:** As indicated by its name, parallel architectures (sometimes also considered as unrolled designs) perform all – or at least a majority – of operations of a (cryptographic) process in parallel and within a single clock cycle. However, due to an increased critical path, hardware designers usually insert pipeline registers into the architecture in order to reduce the critical path and increase the maximum frequency of the design. Hence, parallel implementations usually are optimized for high-performance (i.e., low latency and high throughput) while accepting high area and power demands.

**Serial:** Consequently, serial hardware implementations sacrifice performance at benefit of smaller area and power demands. In particular, recurring operations are only implemented once but executed repeatedly over time. In addition, the internal data path is reduced to a minimum and operations are implemented in fractions, if possible. For modern block ciphers, the nonlinear part, i.e., the S-box usually is the limiting factor and lower bound of the internal data path. Hence, serial architectures usually process a single 8-bit or 4-bit value per clock cycle. Although recent results [JMPS17a] proof the feasibility of bit-serial implementations for modern block ciphers, serialization might not always be an appropriate and unlimited approach for area reduction. Since the operation complexity increases with the degree of serialization, at some point the overhead introduced by additional control logic will cancel all savings, eventually resulting in larger implementations (with lower performance).

**Hybrid:** Naturally, hybrid architectures combine serialization with parallelization strategies, i.e., recurring operations are performed successively whereas all parts of a single operation
are implemented in parallel. Common hybrid architecture strategies for modern block ciphers usually include \textit{round-based}, \textit{column-based}, or \textit{row-based} implementations. During each clock cycle, a single round, column, or row function is applied repeatedly to the internal state which is processed in parallel. Variations in the degree of serialization and parallelization allow hardware designers to adjust the optimization levels with respect to throughput, latency, area, and power consumption. A selection of examples that follows this strategy can be found in [SMTM01, BBR16a, BBR16b].
Chapter 3

Cryptography and Cryptographic Primitives

As a part of this chapter, we briefly discuss modern cryptography and popular cryptographic primitives as basic and fundamental building blocks of our modern security infrastructure. More precisely, we focus on symmetric cryptographic primitives which can be used to provide confidentiality through encryption and decryption. Moreover, we shortly introduce some basic and fundamental concepts of asymmetric cryptography that can be used to construct public-key encryption and more advanced primitives such as digital signature schemes or key agreement protocols.

Contents of this Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Symmetric Cryptography</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>Asymmetric Cryptography</td>
<td>21</td>
</tr>
</tbody>
</table>

3.1 Symmetric Cryptography

Among cryptographic principles, symmetric cryptography is the most popular concept which is already known from ancient times. In general, cryptography has been used in order to protect private communication between two parties against unauthorized listening (eavesdropping) on an insecure communication channel. To this end, both parties share a single, symmetric key in order to encrypt and decrypt messages that are exchanged over the insecure channel. Since the symmetric key is private and only shared between the two communication parties, the concept of symmetric cryptography provides confidentiality for the encrypted messages such that only the authorized communication parties can access the message in plaintext.

**Definition 3.1.1.** A symmetric key encryption scheme is a mapping $E: \mathcal{K} \times \mathcal{P} \rightarrow \mathcal{C}$ with inputs $k \in \mathcal{K}$ (key), $p \in \mathcal{P}$ (plaintext) and output $c \in \mathcal{C}$ (ciphertext). In particular, $E$ has to be invertible, i.e., there exists a second mapping $D: \mathcal{K} \times \mathcal{C} \rightarrow \mathcal{P}$ with $D_k(E_k(p)) = p$, $\forall k, p$.

During the last decades, many modern symmetric-key cryptosystem have been proposed and standardized [Nat99, DPVAR00, DR02, BKL+07, GPPR11, BSS+15]. In general, these systems provide many advantages and have been studied thoroughly by academic and industrial researchers resulting in high confidence on the security of these systems. In particular, the main benefits of secret-key cryptography are that strong symmetric keys can be produced and created easily with significantly smaller key sizes compared to asymmetric keys (cf. Section 3.2) resulting
in encryption schemes that are easy and fast to evaluate even for large amounts of messages or data. Most modern symmetric ciphers that are implemented in practice, can be classified either as stream ciphers or as block ciphers. For both classes, a wide variety of different solution with many advantages and disadvantages have been proposed in recent years. In the following we will discuss the key features of modern stream and block ciphers as well as popular and standardized representatives for both classes.

3.1.1 Stream Ciphers

As one of the first concepts in modern cryptography, the One-Time Pad (OTP) [Ver19] has been presented which provides unconditional or information-theoretic security in theory but suffers from practical problems. In particular, the OTP requires a random one-time pre-shared key for every new plaintext of the same size as the plaintext. Hence, stream ciphers have been introduced as a concept in order to overcome most practical problems but at losing information-theoretic security. In general, modern stream ciphers use a small pre-shared key in order to produce a larger pseudo-random key stream that is then used to encrypt or decrypt the plaintext or ciphertext. To this end, stream ciphers mainly are classified according to their key stream update function. While synchronous stream ciphers use an internal update function for the key stream solely depending on the master key whereas self-synchronizing stream ciphers additionally consider information of the plaintext or ciphertext during the key stream update process.

Within the scope of the eStream ECRTYP Stream Cipher Project [Rob08] running from 2004–2008, a portfolio of modern stream ciphers have been selected for software as well as hardware applications. Among the selections, HC-128 [Wu08], RABBIT [BVZ08], Salsa20/12 [Ber08], and SOSEMANUC [BBC+08] have been considered particularly for software implementations while GRAIN v1 [HJMM08], MICKEY 2.0 [BD08], and TRIVIUM [CP08] have been proposed for hardware applications.

3.1.2 Block Ciphers

Many modern symmetric key encryption schemes, including standards like AES [DR02], are constructed as key-alternating encryption schemes encrypting messages block-wise. In general, a key-alternating encryption scheme uses multiple round functions to compute a ciphertext from an initial plaintext. Each round function uses a different key which is derived from an initial master key using a key scheduling function.

Definition 3.1.2. A key-alternating encryption scheme is a special case for a symmetric-key encryption scheme which is composed of $r$ round functions $E_k^i$ that apply a (pseudo-random) permutation $P^i$ and add a key $k_i$ to the input. The keys are derived from master key $k \in \mathcal{K}$ using a key scheduling function $f : \mathcal{K} \rightarrow \mathcal{K}^r$. The encryption process is constructed as $E_k = E_{k_r} \circ \cdots \circ E_{k_2} \circ E_{k_1}$.

In practice, the round functions of most modern ciphers are either constructed following the principle of a Feistel Network or using a Substitution-Permutation Network. In the following, we briefly highlight the basic notion of both concepts and list common standardized encryption schemes for both classes.
3.2 Asymmetric Cryptography

Feistel Network

The concept of Feistle Networks was introduced by and named after Horst Feistel, one of the designers of the IBM block cipher LUCIFER [Sor84]. In general, Feistel ciphers are key-alternating block ciphers using \( r \) rounds to transform an initial plaintext into an according ciphertext. In particular, for each round the current state of the cipher is split into equal parts \( L_i, R_i \) (i.e., a left and a right half) which are processed such that:

\[
L_{i+1} = R_i \\
R_{i+1} = L_i \oplus F(R_i, k_i)
\]

where \( F \) is the round function that includes the key addition. In particular, Feistel ciphers are well suited for hardware platforms due to their iterative structure and the shared data path for encryption and decryption, i.e., the same architecture can be used for both functions and does not have to be inverted. To this end, the round function \( F \) itself does not have to be invertible and in case it is only consists of addition, rotation and exclusive or this construction often is referred to as Addition-Rotation-XOR (ARX) cipher. Well known Feistel ciphers include LUCIFER [Sor84], DES and TRIPLE DES [Nat99], TWOFISH [SKW+99], RC5 [Riv94], and SIMON and SPECK [BSS+15].

Substitution-Permutation Network

Nowadays, the most popular construction principle for encryption schemes is the Substitution-Permutation Network (SPN) which uses independent substitution and permutation layers within each round function in order to ensure and provide confusion and diffusion according to Shannon’s basic principles [Sha45]. More precisely, the substitution layer usually is composed of small Substitution-boxes (S-boxes) that are applied to only small portions of the internal state (most common S-boxes are 4-bit or 8-bit functions). In combination with the subsequent permutation layer, this structure ensures that small changes in the input of a round results in significant changes of the output (avalanche effect). Depending on the crypt-analytic strength of the chosen substitution and permutation functions, the round function is iterated several times in order to provide an adequate security margin against crypt-analytic attacks such as linear or differential crypt-analysis.

Among the SPN block ciphers, AES [DR02], NOEKEON [DPVAR00], mCRYPTON [LK05], PRESENT [BKL+07], PRINCE [BCG+12], PRIDE [ADK+14], PRINTcIPHER [KLPR10], LED [GPPR11], and ICEBERG [SPR+04] are popular examples.

3.2 Asymmetric Cryptography

The notion of asymmetric cryptography originally has been proposed by W. Diffie, M. Hellman and R. Merkle in 1975 [DH76, Mer78] in order to address problems that cannot be solved solely relying on symmetric cryptographic primitives (e.g., secure key exchange). Instead of establishing a symmetric, shared key between two communicating parties, each user has an asymmetric key pair consisting of a public key, accessible by all parties and solely used for encryption, and a private key, only accessible by its owner and used for decryption. Based on these properties and construction, not only classical encryption schemes, but also digital
signatures and key exchange techniques can be realized. In practice, three different approaches are used in order to build and construct asymmetric cryptographic primitives that are discussed in the following.

### 3.2.1 Integer Factorization Problem

Shortly after the introduction of public-key cryptography, R. Rivest, A. Shamir and L. Adleman introduced the RSA cryptosystem in 1977 [RSA78]. It is based on the assumption and hardness of integer factorization in particular for large integers (nowadays usually of more than 2048 bits). Since most public-key cryptosystems are based on mathematical problems, the RSA system is the most popular example and representative for the problem of integer factorization which, since then, has been applied for many cryptographic protocols.

### 3.2.2 Discrete Logarithm Problem

A second (mathematical) problem which has been used in order to build asymmetric, public-key cryptosystems, is the Discrete Logarithm Problem (DLOG). In general, this problem is based on the assumption that it is hard to find a discrete logarithm in a finite field in adequate time. Consequently, W. Diffie (one of the founder of the concept of public-key cryptography) and M. Hellman introduced a first protocol for key exchange (Diffie-Hellman Key Exchange [DH76]) using the DLOG. Later in 1984, T. ElGamal presented a novel public-key encryption algorithm which is based on the DLOG as well (ElGamal Encryption [Gam84]). Another popular and prominent algorithm based on the problem of finding discrete logarithms, is the Digital Signature Algorithm (DSA) [KG13] which is used to generate and verify digital signatures.

### 3.2.3 Discrete Logarithm Problem over Elliptic Curves

A third family of public-key algorithms is based on a generalization of the DLOG defined on elliptic curves over finite fields. The concept and idea of this approach has been proposed independently by N. Koblitz [Kob87] and V. Miller [Mil85] in the mid 1980s. In general, all algorithms and protocols that have been defined over the DLOG in finite fields (e.g., Diffie-Hellman Key Exchange, DSA, etc.) can be transferred to the discrete logarithm over elliptic curves (e.g., ECDH [LMQ+03], ECDSA [JMV01, ANS05], etc.). Due to its higher complexity, the discrete logarithm over elliptic curves can use smaller parameters while maintaining the same level of security when compared to the DLOG in finite fields or the integer factorization problem.

In particular the reduction in the parameter size led to a rapid (commercial) application that again has been accelerated by the standardization process of ECC. Nowadays, ECC is the predominant standard for high-performance public-key cryptosystems and mainly can be subdivided into three different classes.

#### Elliptic Curves over Prime Fields

Finite prime fields \( \mathbb{F}_p \) consists of all integers \((0, 1, 2, \ldots, p - 1)\) and is constructed by modulo division using a prime number \(p\). Hence, the field has order \(p\) and defines two basic operations (addition and multiplication) modulo \(p\). Using prime number with special structures (e.g.,
Asymmetric Cryptography

Mersenne, Pseudo-Mersenne, Generalized-Mersenne, etc.) can yield in improved and faster reduction algorithms exploiting the underlying structure of the prime. Since modern CPUs usually provide fast and efficient integer multiplication units, many elliptic curves over prime fields, such as NIST P-192, P-224, P-256, P-384, and P-521 [LG09], Curve25519 [Ber06], and Curve448 [Ham15], are favored and optimized for software implementations.

Elliptic Curves over Binary Fields

Binary fields $\mathbb{F}_{2^m}$ are finite fields of order $2^m$ with binary polynomials (i.e., the coefficients are elements of $\mathbb{F}_2 = \{0, 1\}$ of degree $m$ as elements. Addition is defined as simple addition of coefficients of both polynomials and reduction modulo 2 (XOR operation). However, the multiplication uses an irreducible polynomial $i(z)$ and the final results is defined as the remainder of the polynomial division using $i(z)$.

Due to their simple and elegant basic arithmetic (exclusive or and free squaring), elliptic curves over binary fields are in particular favored for efficient and high-performance hardware implementations.

Elliptic Curves over Extension Fields

Extension fields $\mathbb{F}_{p^m}$ are a generalization of prime fields and binary fields whereas $m = 1$ defines the case of prime fields and $p = 2$ leads to the structure of binary fields. Consequently, elements of $\mathbb{F}_{p^m}$ are polynomials with coefficients of $\mathbb{F}_p$ and the basic operations (addition and multiplication) include a reduction using an irreducible polynomial.

Prime extension fields particularly benefit from their flexibility which even can lead to optimal extension fields that are chosen such that $p$ perfectly fits the hardware architecture, i.e., the machine registers and the provided ALU operations. Hence, depending on the targeted CPU architecture, different extension fields can be opted, depending on the features of the underlying system.
Chapter 4

Physical Attacks and Countermeasures

Physical attacks have become the major threat for cryptographic implementations on embedded devices. In particular passive side-channel attacks have been used successfully to mount numerous attacks and disclose sensitive information. Throughout this chapter, we discuss different attacker models, introduce common attacks and evaluation methods of Side-Channel Analysis, and briefly describe basic countermeasure concepts.

Contents of this Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Notations</td>
<td>25</td>
</tr>
<tr>
<td>4.2 Adversary Models</td>
<td>25</td>
</tr>
<tr>
<td>4.3 Side-Channel Analysis</td>
<td>27</td>
</tr>
<tr>
<td>4.4 Evaluation Methods</td>
<td>28</td>
</tr>
<tr>
<td>4.5 Countermeasures</td>
<td>33</td>
</tr>
</tbody>
</table>

4.1 Notations

We denote single-bit random variables using lower-case characters, bold ones for multi-bit vectors, tildes for shared representations, lowering indices for elements within a vector and raising indices for elements of a shared vector.

Furthermore, let us denote any element $x \in \mathbb{GF}(2^m)$ as vector of $m$ single bit elements $(x_1, \ldots, x_m)$. The shared representation $\tilde{x}$ of a vector $x$ using a masking scheme with $s$ shares is given as $\tilde{x} = (x^1, \ldots, x^s)$. Eventually, we denote functions using sans serif fonts and sets using calligraphic ones.

4.2 Adversary Models

In modern cryptography, cryptographic schemes are evaluated against an adversary model in order to assess their security. The classical information-theoretic security model which assumes an unconstrained adversary (in terms of computation power and time) was introduced by Shannon [Sha49]. It has been shown that under this security and adversary model, a secure scheme is achieved if the result does not contain any information on the original input. Nowadays, cryptographic schemes are often evaluated against the complexity-theoretic security model which
assumes an adversary who is bounded in terms of computational power and time. To this end, the following section briefly describes and discusses the general classification of constrained adversaries in this security model.

### 4.2.1 Black-Box Adversary

The most common and traditional adversary model in cryptography assumes and models attackers with strongly limited access to an encryption scheme. In particular, the black-box adversary model only considers access to inputs and outputs of an encryption scheme (excluding the key), but the encryption or decryption process itself is treated as black box (hiding all internal information).

Still, the adversary model defines different level of abilities and limitations for the attacker resulting in several attack scenarios. In a first scenario, a black-box adversary has only passive and observing abilities which results in known-plaintext attacks and ciphertext-only attacks. Considering active but non-adaptive chosen-plaintext attacks and chosen-ciphertext attacks an attacker is able to control inputs of either encryption or decryption processes and observe according results. Finally, an attacker could choose a plaintext or a ciphertext adaptively, depending on previous encryption or decryption results. In particular, this is the most powerful black-box adversary model that can be found in literature and is considered during security evaluation of encryption schemes.

**Definition 4.2.1.** Black-box adversaries have only strongly limited access to an encryption scheme and only are allowed to chose and manipulate inputs and observe corresponding outputs of the encryption or decryption process.

### 4.2.2 Gray-Box Adversary

The most versatile adversary model in cryptography extends the black-box adversary model and defines an attacker that has limited access to an encryption scheme. In particular, the gray-box adversary model still considers access to inputs and outputs of an encryption scheme, but additional information can be retrieved by observing or disturbing the encryption or decryption process. Hence, gray-box adversaries usually do not target the encryption scheme itself but rather practical implementations of the encryption or decryption process. Due to adversarial limitations and practical (i.e., physical) effects, the observed side-channel information is limited and may be hidden within some noise.

Usually, this adversary model defines several attack scenarios with different levels of abilities and limitations for the adversary. First, the gray-box adversary model distinguishes passive attackers that only observe the encryption or decryption process via side channels to gather additional information. In contrast, active attackers are allowed to manipulate and disturb the encryption or decryption process to gather this information. Depending on the appropriated side channels and methods for observing or manipulating the encryption or decryption process, gray-box attacks are considered non-invasive, semi-invasive or invasive.

**Definition 4.2.2.** Gray-box adversaries have limited access to an encryption scheme and can observe or manipulate the encryption or decryption process to retrieve additional information.
4.3 Side-Channel Analysis

4.2.3 White-Box Adversary

The most powerful adversary model in cryptography extends both, the black-box and the gray-box adversary model and defines an attacker that has virtually unlimited access to an encryption scheme. In particular, the white-box adversary model allows the adversary to observe and manipulate every aspect of the encryption or decryption process including inputs and outputs.

Definition 4.2.3. White-box adversaries have virtually unlimited access to an encryption scheme and can observe or manipulate the entire encryption or decryption process.

4.3 Side-Channel Analysis

Since its discovery by Paul C. Kocher [Koc96], SCA has become the most prominent and well-studied type of passive gray-box attacks. Based on physical observations, an adversary can recover secret information from side-channel leakage by means of visual or statistical examination of the measurements.

4.3.1 Physical Side Channels

Sensitive and secret information processed within embedded devices can leak through various physical channels. In the following, we briefly introduce the most common physical side channels that are considered in open literature.

Execution Time

Straightforward and inconsiderate implementations of cryptographic algorithms may have time-variant characteristics depending on processed data. Providing that execution time differences depend particularly on secret or sensitive data, exact measurements of the timing behavior enables to reveal sensitive information using side-channel analysis. In practice, software-based or sequential implementations are particularly prone to timing attacks, e.g., due to conditional branches based on sensitive data, memory and cache accesses, arithmetic operations, etc. Ever since the seminal work of Kocher [Koc96], many successful attacks have been presented eventually leading to the knowledge of considering timing characteristics during the design process and finally resulting in many time-invariant cryptographic implementations.

Power Consumption

Modern electronic devices and circuits, most commonly manufactured based on CMOS technology, mostly have a dynamic power consumption\(^1\) depending on processed data. In particular, internal signal switches and transitions of transistors and gates in a CMOS circuit create an increased current which is drawn in order to charge internal capacitances. Along with some static part \(P_{static}\), the total power consumption over time can be expressed as:

\(1\)Usually, the voltage drop over a resistor \(R\) is measured which technically is proportional to the consumed current \(I_{CC}\) (according to Ohm’s law: \(U_R = R \cdot I_{CC}\)). Although current consumption would be the correct expression, power consumption has been established as common term in the field of SCA research. In that sense, we will stick with this convention and use power consumption interchangeably with current consumption.
\[ P(t) = P_{\text{static}} + P_{\text{dyn}}(t), \quad (4.1) \]

where \( P_{\text{dyn}}(t) \) is the fraction caused by signal switches and transitions. However, due to physical phenomena and interferences, practical measurements are far from ideal and usually have an additional noise term (to consider measurement and algorithmic noise), eventually yielding:

\[ P_{\text{noisy}}(t) = P_{\text{static}} + P_{\text{dyn}}(t) + N(\mu, \sigma^2) \quad (4.2) \]

with the noise modeled by a normally distributed random variable \( X = N(\mu, \sigma^2) \) with \( \mu \) and \( \sigma^2 \) as the mean and variance. Consequently, the instantaneous power consumption of a cryptographic device provides information on (sensitive) internals through its dynamic power consumption. More precisely, although the information may be obscured by measurement or algorithmic noise, the information still can be extracted as initially shown by Kocher et al. [KJJ99] followed by many more.

**Electromagnetic Emanations**

As a consequence of current flowing within an electronic circuit, electromagnetic (EM) emanations in terms of changes in the electric field \( E \) and magnetic field \( H \) arise. While intentional emanations are directly caused by current flowing within a device, unintentional EM emanations arise mainly due to the general circuit layout, coupling effects and other interferences. In general, both types of emanations strongly depend on the internal current flow (i.e., the power consumption) of a cryptographic device and provide information on (sensitive) internals through its dynamic behavior. However, in comparison to rather shallow power analysis, EM analysis is far more powerful and potent. In particular, precise probing and control of spatial parameters during analysis allows filtering of unnecessary information giving a strong focus on sensitive parts and areas of a cryptographic circuit. Still, in many cases coarse and loose information on the current flow within a device (based on its total power consumption) are sufficient to extract sensitive data.

**Other Side Channels**

Apart from classical and well studied side channels (i.e., execution time, power consumption, electromagnetic emanations), various other physical characteristics have been considered and proposed as side channels. Among others, acoustic noise [GST14, GST17], temperature variations [BKMN09, HS13], or photonic emissions [SNK+13, KNSS13, CSW16, CSW17] have been studied and shown as a viable source of side-channel leakage. However, due to strong relations of underlying physical properties or complex measurement setups, classical side channels still attract most attention of academic and industrial communities and are the focus of this work.

**4.4 Evaluation Methods**

Security evaluation of embedded devices, in particular during its development process, is a challenging task. While early side-channel research focused on attack-based evaluation strategies,
alternative solutions have been proposed and considered in recent years. More precisely, due to an ever increasing number of different attacks and more complex countermeasures, attack-based cannot provide a comprehensive security analysis. In this context, test-based and information-theoretic approaches provide an efficient and generic framework for evaluation of embedded devices in terms of side-channel security. The following section briefly reviews basic attack-based methods of SCA before introducing the most popular test-based evaluation strategy for statistical leakage assessment.

4.4.1 Direct Observation

Given only a single measurement of a cryptographic operation or the average over multiple measurements with constant input (in order to reduce the noise), simple analysis relies on direct interpretation of the traces to disclose sensitive information of the performed cryptographic operation. Usually, the inspection process requires deep knowledge of the target device and implementation since it requires the detection – most often visually – and interpretation of data-dependent patterns in the measurement traces.

4.4.2 Differential Analysis

In contrast to Simple Analysis, Differential Analysis usually does not require deep knowledge of the implementation under analysis. Instead, a leakage model is used in order to model physical characteristics and the behavior of a device under test. Then, by means of different statistical methods and distinguishers, hypothetical internal states are tested and evaluated against the physical observations based on the chosen leakage model in order to deduce processed sensitive information.

**Difference of Means**

This basic statistical side-channel distinguisher, already presented in 1999 by Kocher et al. [KJJ99], is based on a most simplistic and abstract leakage model (only assuming detectable differences for single bits taking values of 0 or 1). More precisely, it uses a divide-and-conquer approach to predict a part (i.e., a single bit $v_i$) of a sensitive intermediate value within a cryptographic device. Given exploitable side-channel leakage of the intermediate value and a sufficient number of side-channel measurements $X_i$ arranged in two sets $Q_{v_i=0}, Q_{v_i=1}$ according to the predicted hypothetical intermediate value, the correct guess of the sensitive value can be revealed due to a significant difference $\delta$ in the mean of both sets (for the correct point in time). In particular, for every sample point in time, the distinguisher is derived as:

$$\delta = \mu_0 - \mu_1$$  \hspace{1cm} (4.3)

with $\mu_0, \mu_1$ the mean of the sets $Q_{v_i=0}, Q_{v_i=1}$. In case the predicted hypothetical value is incorrect, the assignment of measurements to the sets is assumed to be random yielding virtually identical means for both sets. However, for a correct prediction the leakage in both sets is significantly different which eventually results in distinguishable means for both sets. Hence, for the correct point in time (corresponding to the processing of the sensitive information) the difference of means $\delta$ shows the highest absolute value verifying the correct guess.
Chapter 4 Physical Attacks and Countermeasures

Pearson Correlation Coefficient

Correlation-based analysis has been introduced to cope with enhanced and more complex power models (compared to the single bit model) which provide better simulations of physical properties and processes of cryptographic devices responsible for side-channel leakage. More precisely, given an adequate power model $f$, e.g., the Hamming Weight (HW) or Hamming Distance (HD) model, each hypothetical intermediate value $v_i$ is mapped to $y_i = f(v_i)$ in order to model the leakage process. In order to distinguish the correct guess of the sensitive intermediate value, the (sample) Pearson correlation coefficient $r_{xy}$ is derived as:

$$r_{xy} = \frac{\text{cov}(X,Y)}{\sqrt{\text{var}(X)} \sqrt{\text{var}(Y)}} = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n \sum x_i^2 - (\sum x_i)^2} \sqrt{n \sum y_i^2 - (\sum y_i)^2}}$$

(4.4)

with $x_i, y_i$ single samples from the physically measured traces $X$ respectively the power model predictions $Y$ and $n = |X| = |Y|$ the number of samples. For correct guesses of intermediate values, the correlation coefficient will indicate a linear dependency of $X$ and $Y$ by $r_{xy}$ being significantly closer to $+1$ (positive linear correlation) or $-1$ (negative linear correlation) than for wrong guesses. Again, this analysis is performed for all sample points in time independently and only samples corresponding to the processing of sensitive information (i.e., the correct points in time) will indicate the correlation.

4.4.3 Statistical Leakage Assessment

In recent years, several initiatives and attempts to define a standardized approach for leakage assessment and side-channel evaluation of cryptographic hardware have been launched. As a result, the statistical leakage assessment based on the renowned Welsh’s $t$-test has been proposed by Goodwill et al. [GJJR11] and has been widely deployed since then. In particular, the non-specific version of the Test Vector Leakage Assessment (TVLA) offers a fast, reliable, robust, and generic evaluation approach to detect and assess potential leakage of a cryptographic device. The following section provides a brief summary of the statistical background before giving a detailed description of the TVLA methodology.

Statistical Background

In the context of sampling from a given population with unknown probability density function, using statistical moments for approximation of the distribution is a more efficient approach than using histograms or kernels. Since the basic concept of the leakage assessment methodology employed in this work relies on the decision whether two samples were drawn from the same population or not, comparing statistical moments can provide necessary information. Hence, for an easier understanding, we first introduce the notion of statistical moments in order to characterize and describe distributions before discussing the leakage assessment methodology.

Statistical Moments. Moments about the origin of a random variable $X$ are considered as raw moments where $M_1$ denotes the first-order raw statistical moment corresponding to the mean $\mu$ of the source $X$. 

30
4.4 Evaluation Methods

Definition 4.4.1. The $d$-th order raw moment $M_d$ is the expected value of $X^d$:

$$M_d = E(X^d). \quad (4.5)$$

Further, moments about the mean $\mu$ of a random variable $X$ are referred to as central moments where $CM_1 = 0$ and $CM_2$ is the second-order central moment denoted as variance $\sigma^2$ of $X$.

Definition 4.4.2. The $d$-th order central moment $CM_d$ is the expected value of $(X - \mu)^d$:

$$CM_d = E ((X - \mu)^d), \quad (4.6)$$

Eventually, moments with respect to the mean $\mu$ and standard deviation $\sigma$ denote normalized moments (sometimes also referred to as standardized moments) where $NM_1 = 0$ and $NM_2 = 1$. Further, $NM_3$ is the third-order normalized moment specified as skewness $\gamma$ and $NM_4$ is the fourth-order normalized moment corresponding to the kurtosis $\kappa$.

Definition 4.4.3. The $d$-th order normalized moment $NM_d$ is the expected value of $(X - \mu)^d$ normalized by $\sigma^d$:

$$NM_d = E \left( \frac{(X - \mu)^d}{\sigma^d} \right), \quad (4.7)$$

General Normal Distribution. Due to the occurrence of physical noise in side-channel measurements, a general normal (sometimes also considered as Gaussian) distribution is the common model to describe and define characteristics of the measurements. The probability density function of this distribution is defined as:

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (4.8)$$

where $\mu$ and $\sigma^2$ are the first and second statistical moment (mean and variance), and $f$ returns the relative likelihood of a random variable $X$ to take a specific value $x$. Since the general normal distribution depends only on the first two statistical moments, it has a constant skewness ($\gamma = 0$) and kurtosis ($\kappa = 3$). Therefore, in order to take higher statistical moments into account during side-channel analysis, different distributions have to be used for higher-order attacks and evaluations.

Test Vector Leakage Assessment (TVLA)

In 2011, the TVLA methodology has been proposed by Cryptography Research Inc. [GJJR11] as a robust and reliable assessment approach of the physical vulnerabilities of cryptographic devices. In that sense, TVLA mainly focuses on detection of side-channel leakage (without any assumption on its nature) rather than its exploitation, e.g., for key recovery.

In general, TVLA is an empirical strategy based on physical measurements used to detect side-channel leakage. The basic assumption of this methodology is that if we can distinguish two sets (given these two sets of measurements are different by construction), this is tantamount to side-channel leakage or additional information detectable in the measurements. In other words, if the null hypothesis as the samples of both sets were drawn from the same population is rejected with adequate confidence, side-channel leakage is assumed within the measurements. Consequently, the TVLA methodology defines necessary techniques to construct two sets and evaluate the measurements against the null hypothesis.
Distinguishing Criteria. Precursory to the null hypothesis testing and leakage detection, physical measurements of a Device Under Test (DUT) in operation are performed with respect to a chosen distinguishing characteristic in order to construct two different sets of measurement data. During the observation and evaluation process, the device is queried with associated data while a chosen side channel is observed over time and measurements are stored as traces of time-independent sample points. Since the secret key of the cryptographic DUT is constant during the evaluation process and usually known to the evaluator, several opportunities to construct different sets of measurement data and to control the granularity of evaluation process arise:

Specific Criterion: Given knowledge of the secret key and any associated data, all intermediate values of the cryptographic operation on the DUT can be reconstructed by the evaluator. Based on a specific sensitive intermediate value, parts of it (e.g., single bits, bytes, etc.), or even the combination of multiple sensitive values, the recorded traces can be grouped into different sets and evaluated against the null hypothesis on a very fine level. Common specific criteria include, i.e., the bit model or byte-wise classifications in order to compare the sets and evaluate the null hypothesis. However, this approach still suffers from the same weaknesses as state-of-the-art side-channel attacks due to its fine granularity since it implies a plethora of different models to test (i.e., sensitive intermediate values) which eventually prevents a comprehensive evaluation.

Semi-Specific Criterion: In order to alleviate the drawbacks of specific distinguishing criteria, semi-specific evaluation methods increase the degree of freedom and granularity for the testing model and consequently reduce the evaluation process complexity. Still, recorded traces are grouped according to intermediate values of certain structure, however, a second set is chosen at random. Hence, while the first set contains measurements with associated data that leads to intermediate values with similar patterns (e.g., certain numbers of leading zeros), the second set only includes measurements with randomly chosen associated data. Although this approach compares specific to non-specific sensitive values on a more coarse-grain level, it only reduces the dependencies on underlying testing models but still has a nearly unmanageable complexity.

Non-Specific Criterion: Consequently, low complexity and maximum independence of testing models is the ultimate objective of any evaluation methodology. Hence, constructing a set of repeated measurements with fixed associated data and comparing this set to random measurements is considered as non-specific evaluation criterion. Since for fixed associated data, all sensitive intermediate values are constant and compared to random data, the precision of the evaluation coverage zone is increased to a maximum, resulting in a generic evaluation process. Particularly, the construction of both sets only depends on the associated data and does not require exact knowledge of intermediate values or even secret information. In practice, most evaluation procedures use either plaintexts or keys for fix versus random associated data and the non-specific evaluation criterion.

Hypothesis Testing. In TVLA, testing the null hypothesis as two population have equal means (in other words, both populations are supposed to be identical and all samples were drawn from the same population), is based on Welch’s t-test which is an adaption of Student’s (two-tailed) t-test particularly designed for robust testing using sampled sets with unequal variances and
4.5 Countermeasures

different set sizes. Given two sets \( Q_0, Q_1 \), with \( n_0, n_1 \) the cardinality of both sets and let \( \mu_0, \mu_1 \) and \( \sigma^2_0, \sigma^2_1 \) be the sample mean and variance of both sets, the \( t \)-statistics is defined as:

\[
t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{\sigma^2_0}{n_0} + \frac{\sigma^2_1}{n_1}}}
\] (4.9)

Given the two sets \( Q_0, Q_1 \) of recorded traces, the evaluation process is performed in an univariate fashion\(^2\), i.e., on each sampled point in time individually. Assuming a general normal distribution of the sampled points, the \( t \)-statistics give evidence of differences in the first statistical moment (mean) of both sets, hence it can detect first-order side-channel leakage. In order to evaluate the measurements against higher-order leakages, the recorded traces have to be pre-processed with respect to higher-order statistical moments before the mean of both sets can be estimated and the \( t \)-statistic can give evidence of differences and potential vulnerabilities of the DUT. In particular, for a second-order, univariate, mean-free squared traces \( Y = (X - \mu)^2 \) (i.e., the variance \( CM_2 \) of the unprocessed traces) are evaluated based on the expected value of \( Y \) (mean) using the \( t \)-statistic. Similarly, higher-order, univariate evaluations take the \( d \)-th order normalized statistical moment \( NM_d \) of the traces into account.

**Leakage Assessment.** Eventually, in order to accept or reject the null hypothesis with adequate confidence, sound assessment of the evaluation results and potential side-channel leakage is necessary. For the sake of simplicity and usability, the absolute value of the \( t \)-statistic is compared to a predefined threshold of 4\(^5\) during the final assessment process. This simplification is based on the fact that, for sufficient samples (usually more than 1000), absolute \( t \)-values of more than 4\(^5\) allow to reject the null hypothesis with a confidence of more than 0.99999.

4.5 Countermeasures

Along with improved methods of SCA, the development of efficient and secure countermeasures against physical attacks is an essential aspect of recent research efforts. In the course of time, three different concepts to design efficient countermeasures have emerged: masking, hiding, and rekeying. In particular, masking and hiding have been well-studied and analyzed by academia throughout recent years. The following section briefly summarizes their underlying notions, before presenting prominent and widely used solutions for both concepts.

4.5.1 Hiding

Most side-channel leakage detection and exploitation mechanisms are susceptible to random noise within the physical measurements, i.e., with increasing noise level the success rate of leakage detectability or exploitability diminishes. Consequently, increasing the noise in order to hide potential side-channel leakage is an obvious approach to counteract physical vulnerabilities.

\(^2\)In order to take potential multivariate leakages into account, the traces and sample points have to be preprocessed with respect to mixed statistical moments. Since evaluation of multivariate leakages using TVLA is beyond the scope this work, we refer the interested reader to [SM15].
of cryptographic devices. In particular, the SNR, defined over the variances of the leakage and noise signals, as:

$$SNR = \frac{\text{var}(\text{signal})}{\text{var}(\text{noise})}$$ (4.10)

is a common metric to illustrate and assess the concept of hiding countermeasures. In general, decreasing the SNR, either by reducing the leakage signal or by amplifying the measurement noise, helps to improve the level of protection of a cryptographic design although it does not provide provable security. More precisely, hiding countermeasures cannot provide absolute protection but only harden a device practically and usually have to be complemented by additional countermeasures. To this effect, the devices still remain vulnerable to side-channel attacks (at any order) but adversaries have to spend a lot more effort in terms of measurement time and computational power to disclose secret information.

**Leakage Signal Reduction**

Due to its excellent properties in terms of low static power consumption, CMOS is the predominant technology for modern (cryptographic) hardware devices. However, since side-channel leakage is mainly caused by dynamic behavior of a hardware architecture, CMOS circuits are particularly prone to side-channel analysis based on power consumption and electromagnetic radiation. Hence, most signal reducing hiding countermeasures are hardware-based and act on circuit and gate level while optimizing power consumption and reducing data dependencies. In general, these custom logic styles try to render the leakage independent of the process data and operations, e.g., by equalizing the power consumption using differential (dual-rail) signal encodings and precharge logic styles \[TAV02, TV04, CZ06, PM05, PKZM07\], however usually at cost of always obtaining the worst case scenario (i.e., high power consumption).

**Noise Amplification**

Instead of reducing the signal and leakage information, amplification of the physical noise in most cases is a simpler and more practical solution. In general, noise amplification can work in different ways, i.e., either acting in the amplitude or in the time dimension of the measured traces, essentially dealing with different sources of (random) noise.

**Amplitude Dimension.** Dynamic behavior of cryptographic devices that is unrelated to sensitive intermediate values and information is commonly considered as (random) arithmetic noise within the measurements. Hence, any additional hardware component that is leaking information independent of sensitive values serves as noise generator. More precisely, implementing additional components, e.g., to increase the dynamic power consumption of a CMOS circuit intentionally, changes the amplitude of the measurements due to amplification of the noise level by which side-channel analysis eventually becomes increasingly difficult.

**Time Dimension.** In particular for cryptographic software implementations, random insertion of dummy operations and shuffling of the execution order are popular approaches aiming for noise amplification and hiding in the time domain. For hardware-based implementation, trace
misalignment due to clock signal randomization [GM11], produces similar effects. In general, since the leakage of sensitive information is shifted and spread over time and occurs at different points of the recorded traces, samples corresponding to unrelated operations or data will contribute as (random) noise to the measurements. Consequently, statistical analysis of physical side-channel measurements, which is usually vulnerable to noise, becomes harder.

### 4.5.2 Masking

Randomization of processed sensitive intermediate values of cryptographic devices is a different approach to prevent side-channel leakage, commonly known as masking, which has been studied extensively by the academic community [CJRR99]. Striking with provable security due to its sound theoretical foundation, it is a versatile approach which can prevent side-channel leakage at various levels (e.g., algorithmic-level or gate-level). Following the concept of secret sharing, sensitive variables $x$ are split up into $s$ shares $x^i$ ($1 \leq i \leq s$) such that:

$$x = x^1 \circ x^2 \circ \cdots \circ x^s$$

(4.11)

with $\circ$ the group operation defined by the particular masking scheme. In that sense, $x$ is concealed by $s - 1$ random masks $x^i$ ($1 \leq i \leq s - 1$) with $x^s$ chosen as a function of $x$ and all random masks such that Equation 4.11 is satisfied. Note that the level of protection and security increases with the number of shares.

#### Boolean Masking

Boolean masking, probably the most frequently deployed masking scheme, relies on Exclusive OR (XOR) as the group operation. In particular, a first-order Boolean masking scheme is constructed as:

$$x_1 = m$$
$$x_2 = x \oplus m.$$ 

(4.12)

Given that all computations on the shares leak independently and $m$ is chosen randomly, only the joint distribution of both shares leaks information on $x$ resulting in a first-order secure masking scheme. Similarly, Boolean masking schemes using up to $d + 1$ shares (assuming that all $d$ random masks are independent) can provide provable security with respect to $d$-th order attacks.

#### Threshold Implementation

TI is a widely used technique to protect hardware devices against physical attacks. In particular, TI is based on Boolean masking and multi-party computation and provides provable security, even in the presence of glitches\(^3\). In general, any TI has to provide the following three properties:

\(^3\)For a more detailed description, please refer to the original articles [NRR06, NRS11, BGN+14b].
Correctness. Given a vector $\vec{x} = (x^1, \ldots, x^s)$ in its shared representation, we can compute any function $F(\vec{x}) = \vec{y}$ on it but have to ensure correctness, i.e., the result $\vec{y} = (y^1, \ldots, y^t)$ has to be shared representation of $y = F(x)$ with $t \geq s$. But for this purpose, we can use according component functions $f^i$ to evaluate $F$ for each share individually. However, finding correct component functions is not trivial, in particular if $F$ is a nonlinear function [BB16]. In addition, each component function has to ensure further properties such as non-completeness and uniformity.

Non-Completeness. As mentioned before, each resulting share $(y^1, \ldots, y^t)$ is given by an individual evaluation of a component function $f^i \in \{1, \ldots, t\}(\cdot)$ over the input shares. However, in order to achieve security in sense of first-order statistical moments, each component function has to provide non-completeness. This means that each component function $f^i \in \{1, \ldots, t\}(\cdot)$ must be non-complete, i.e., independent of at least one input share.

Uniformity. The security of TIs as masking schemes is based on the uniform distribution of the mask respectively the shared representation which serve as input for a function evaluation. However, since results of a function, e.g., an S-box, are used as input to another function, the outputs of the functions again have to be uniformly distributed. This means, given a set of all possible input sharings $X = \{\vec{x} | \bigoplus_{i=1}^s \vec{x} = \vec{x}\}$ the set of all possible output sharings, i.e., $\{(f^1(\cdot), \ldots, f^t(\cdot))|\vec{x} \in X\}$ should be drawn uniformly from the set of $Y = \{\vec{y} | \bigoplus_{i=1}^t \vec{y} = \vec{y}\}$ as all possible sharings of $y = F(x)$.

Higher-Order Threshold Implementation

An extension of the first-order secure TI concept in order to provide higher-order security has been proposed in [BGN+14b] and improved in [RBN+15]. Although the general concept and the basic properties are almost similar, the HO-TI scheme has a different requirement for the non-completeness property:

Higher-Order Non-Completeness. Again, each resulting share $(y^1, \ldots, y^t)$ is derived by an individual evaluation of each component function $f^i \in \{1, \ldots, t\}(\cdot)$ over the input shares $(\vec{x}^1, \ldots, \vec{x}^t)$. For $d$-th order protection, however, in the sense of security up to the $d$-th statistical moment, each component function has to provide $d$-th order non-completeness. This means that each combination of up to $d$ component functions must be non-complete, i.e., independent of at least one input share.
Part II

Efficient Implementations of Cryptographic Primitives
Chapter 5

Side-Channel Protected Low-Area AES Implementation

Block ciphers are among the most important cryptographic primitives in practical applications and AES certainly is the predominant representative used worldwide in many cryptographic applications. In this chapter, we introduce three different low-area implementations of AES-128 particularly tailored for reconfigurable hardware based on a joint work with Tim Güneysu [SG16a]. First, our basic proposal presents a minimal but key-agile AES encryption architecture that yields the smallest implementation reported in open literature, occupying just 21 slices of a Spartan-6 and no additional memory. Second, we show that the basic architecture almost inherently supports side-channel countermeasures at minimal extra costs but still fits into 24 slices. Third, we present a protected, self-contained but lightweight AES-128 encryption core that combines previous results with random number generation within 28 slices. Finally, we analyze and evaluate the security of our final design practically under SCA using CPA and show that our approach increases the resistance against CPA attacks significantly while maintaining lightweight properties.

Contents of this Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>39</td>
</tr>
<tr>
<td>5.2</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>42</td>
</tr>
<tr>
<td>5.4</td>
<td>48</td>
</tr>
<tr>
<td>5.5</td>
<td>49</td>
</tr>
<tr>
<td>5.6</td>
<td>51</td>
</tr>
</tbody>
</table>

5.1 Introduction

Today, AES is the predominant choice of standardized cryptography for virtually any device that has a need for data confidentiality, authentication or both. Since its standardization, a lot of hardware implementations have been proposed spanning over a wide range of design rationales and optimization goals, such as high-throughput, low-area, low-power or low-energy implementations. Although a high-performance cryptographic engine providing bulk data encryption is
essential for many security related functions, in some cases the situation is somewhat different. Although the AES has been carefully designed to provide a high security guarantee at reasonable implementation cost in software and hardware, it still remains a rather expensive solution when being deployed in devices designed for lowest costs (e.g., one of the smallest, physically unprotected ASIC implementation requires 2400 GE [MPL+11]). This fact was also one of the reasons for strong research interest in lightweight cryptography, giving rise to a wealth of other very small block ciphers, such as PRESENT [BKL+07], NOEKEON [DPAR00], PRINCE [BCG+12], or SIMON and SPECK [BSS+13].

Nevertheless, the situation remains unsatisfying when designers are faced with the strict requirement to integrate standardized cryptography (and such AES) in their system which is, in the worst case, even required by multiple components within a complex SoC. Moreover, since many of those components within the SoC architecture are provided by different vendors with no mutually trusted relationships, it is not uncommon that every security-critical SoC component has a separate security subsystem with an individual AES core. In particular, for advanced security functions such as feature activation and IP licensing of components some schemes may even require per-component availability of an individual AES [DK09].

If the AES core is only considered a rarely used and supplementary function, it is essential to keep its resource consumption as low as possible. But though being a low-cost supplementary function, aspects of physical security of an AES implementation in hardware still need to be considered. Any cryptographic hardware implementation that is potentially physically exposed to an attacker must include countermeasures against SCA attacks [MOP07]. Most protection mechanisms counteracting SCA require randomness to mask or hide the exploitable side-channel information, hence this demands for the availability of a connected and potent Random Number Generator (RNG). Block ciphers can be used as Cryptographic Pseudo Random Number Generators (CPRNGs) in order to expand random seeds obtained from an entropy source. In this sense, all supplementary cryptographic implementations should be lightweight, physically protected and as far as possible self-contained and include a facility to generate the required randomness e.g., for key generation or to feed physical countermeasures.

5.1.1 Related work

We briefly summarize previous results of relevance to this work and our contribution. Since there is already a wealth of publications addressing AES architectures on reconfigurable hardware, e.g., [CG03, JTS03, SRQL03, RSQL04, HV04, GB05, CKVS06, DGP08, BSQ+08], we restrict the discussion of previous works to the most relevant ones. As one of the first AES implementations optimized for resource-limited applications on reconfigurable hardware, in 2005, Good and Benaissa [GB05] proposed an Application-Specific Instruction Processor (ASIP) based on Xilinx’ PicoBlaze [Xil11] soft-core microcontroller. This publication was followed by many more, but one of the first architectures that was designed for the newer FPGA generations starting with Virtex-5 is given in [BSQ+08], where for the first time a minimal S-box implementation of 8 slices was introduced. In 2012, Chu et al. [CB12] picked up the idea of Chodowiec and Gaj [CG03] to use Shift Register Look-Up Tables (SRLs) in order to store and process intermediate values during the round computation. For their final design, the authors can report a size of 80 slices for an encryption architecture on a Spartan-6 FPGA, which is the smallest memory-free AES-128 FPGA implementation to date.
5.1.2 Contribution

In this chapter, we present a threefold contribution. First we propose an extremely lightweight AES-128 implementation optimized for recent Xilinx FPGA generations (starting with Virtex-5). To the best of our knowledge, our proposal is the smallest AES-128 design reported so far and fits into 21 slices of a Spartan-6 with no additional BRAM. Our approach takes solely advantage of the Distributed Memory instances of recent Xilinx FPGAs that can be perfectly matched with an 8-bit serialized AES architecture. Second, we show that our basic concept can be easily enhanced to provide an AES-based CPRNG that can be, finally, used to build a side-channel protected, self-contained AES-128 encryption architecture occupying not more than 28 slices. Despite the minimal resource requirements, a single encryption with our self-contained, protected AES-128 core takes 1471 cycles that translates to the throughput of 4.35 Mbps. We also want to emphasize that all of our proposed designs include the key schedule and hence are fully functional in order to perform AES encryptions even with frequent rekeying. Eventually, we also performed practical side-channel evaluations of our self-contained AES-128 encryption architecture and show its significantly increased resistance against physical attacks while still maintaining its lightweight and low-area properties.

Note that our compact encryption architecture easily outperforms any PicoBlaze [Xil11] instance (an 8-bit soft-core microcontroller tailored for Xilinx FPGAs) that runs a software AES. Such an instance requires 26 slices and additional BRAM in its minimal configuration and more than 13546 cycles [GB05] per encryption.

5.2 Design Considerations

Usually, lightweight cryptographic architectures are constrained by the storage of intermediate states and the width of their internal data path. Fortunately, AES mainly relies on byte-wise operations and inherently supports an area efficient 8-bit data path. Besides, modern FPGAs are typically rich in memory elements, in particular considering Distributed Memory primitives. Observing that modern Xilinx FPGAs offer up to 256-bit Distributed Memory per slice yields the opportunity to push the limits for resource-constrained implementations on FPGAs further by reducing the area of state registers. Moreover, we could identify the SubBytes and MixColumns operations as most area and logic consuming functions that require some special effort in order to implement low-area instances. For the S-box, the smallest implementation found in the literature (based on 6-input LUTs), still requires at minimum 8 slices which leaves no room for optimization. However, the MixColumns operation mainly applies byte-level Galois Field (GF) arithmetic (although operating on 32-bit columns) and can be merged easily with the key addition operation. By this, the total round function logic and its critical path finally could be reduced to the S-box and a subsequent MixColumns unit. Since each round requires different sub-keys that have to be derived on the fly, a complete AES-128 encryption core requires additional logic for key expansion. Fortunately, the key update unit can be merged entirely with the round function logic since only a few byte-wise substitutions using the S-box as well as Galois Field (GF) arithmetic (provided by the MixColumns unit) are required to derive a new sub-key. Thus, by sharing resources we can implement an ultra-lightweight, low-area AES-128 encryption core including key expansion in a handful of slices.
5.2.1 Protection against Physical Attacks

Although designed to be lightweight and low-area, our design should include countermeasures to thwart physical side-channel attacks. We therefore employ a design methodology that easily allows shuffling of the byte execution order as hiding-based countermeasure. In particular, the individual access to stored bytes (thanks to the utilization of Distributed Memory) provides the option to extend our design quite naturally with this countermeasure.

5.2.2 Pseudo Random Number Generation

Commonly, True Random Number Generator (TRNG) are unable to provide sufficient randomness required by many side-channel countermeasures in very short time. Hence, in practice a hybrid approach is used, where a TRNG can provide some random seed which then is expanded using a subsequent Pseudo Random Number Generator (PRNG). Often, block cipher primitives are used to extract the pseudo-randomness from an entropy source since their results provide all necessary statistical properties, then usually considered as CPRNG. Our self-contained encryption engine offers straightforward protection against side-channel attacks and is independent of external RNGs (after an initial random seed has been provided) since encryption runs are interleaved with the generation process of pseudo randomness using the AES core.

5.2.3 Modes of Operation and Decryption

Our architecture is designed to support encryption but no decryption operation. However, in practice encryption is most of the time used in a specific mode of operation, i.e., the encryption core is embedded into a particular configuration that satisfies application-specific requirements, such as block-wise, parallel encryption. An extremely popular mode of operation is the Counter Mode (CTR), which turns the original block cipher into a stream cipher. The advantage of this mode is that both encryption and decryption can be achieved by using an encryption-only core that generates blocks of a key stream that are added to the plaintext and ciphertext, respectively. The key stream is generated using a random nonce concatenated with a counter that is incremented for every block and the encrypted. Taking this into account we assume that an encryption-only core should be sufficient for most applications with need for lightweight cryptography.

5.3 Implementation

In this section, we present details of our basic encryption architecture. Although this implementation is designed for the Xilinx Spartan-6 FPGA family, it can be easily ported to any other FPGA device family providing 6-input LUTs and a similar integrated memory option. By aggressive sharing of components we limit the resource requirements of the basic implementation to 15 slices for all functional blocks and additional 6 slices are necessary to control the entire encryption process. In a second step, we extend our basic core design with a fundamental protection feature against side-channel analysis by introducing random shuffling of the byte execution order during round computations. We like to highlight that due to the architecture of our basic core this is possible by only slightly modifying the architecture. All required randomness to derive a permutation of the execution order can be provided either by an (additional)
5.3 Implementation

Figure 5.1: Architecture overview of the basic AES-128 encryption core: The round function and key schedule are implemented in 15 slices, the remaining 6 slices are necessary for control logic (omitted in the figure for the sake of clarity).

external RNG or even by the AES-128 encryption core itself that is employed as CPRNG after being provided with an initial (random) seed. We remark that even with a CPRNG option, the introduced overhead is still minimal and the resource requirement of the basic core is only slightly increased.

5.3.1 Basic Encryption Core

The overall architecture of our AES-128 encryption and key update function is shown in Figure 5.1. All state and round key information is stored within two Distributed Memory components.

Initialization

Prior to each encryption, registers (Figure 5.1: STATE and KEY) are initialized with the plaintext and key, respectively. Both registers are implemented as 256-bit Distributed Memory (configured as 32 × 8-bit memory primitives). Plaintext and key bytes have to be provided alternately, for two clock cycles each. While all bytes of the key are stored unaltered, the round state register is initialized with the result of the initial key-whitening operation. In total, the initialization phase for a new encryption requires 65 clock cycles.

Key Schedule

Before a single round of an AES encryption can be executed, the key update function is applied to the key register state, in order to compute the sub-key. This process is performed in a byte-sequential flow of 50 clock cycles merged with the round function logic and requires 4 S-box computations and 17 Galois Field (GF) operations (one constant and 16 byte additions). Unfortunately, due to the computation using byte-wise operations, data dependencies are introduced which restrict the degree of flexibility for this process (with respect of shuffling the
Algorithm 1: Byte-sequential key expansion of AES-128.

**Input:** \(K^0 = k_0^0, ..., k_{15}^0, RC = r_{c0}^0, ..., r_{c10}^0\)

**Output:** \(K^1 = k_0^1, ..., k_{15}^1, K^2, ..., K^{10}\)

for \(i \in \{0, ..., 9\}\) do

\[
\begin{align*}
    k_{0}^{i+1} &= k_{0}^{i} \oplus S(k_{13}^{i}) \oplus r_{c_{i+1}} \\
    k_{1}^{i+1} &= k_{1}^{i} \oplus S(k_{14}^{i}) \\
    k_{2}^{i+1} &= k_{2}^{i} \oplus S(k_{15}^{i}) \\
    k_{3}^{i+1} &= k_{3}^{i} \oplus S(k_{12}^{i})
\end{align*}
\]

for \(j \in \{4, ..., 15\}\) do

\[
    k_{j}^{i+1} = k_{j}^{i} \oplus k_{j-4}^{i+1}
\]

end

end

execution order). However, since the round key occupies only half of the register, the second half is used to store all round constants.

Algorithm 1 shows the byte-sequential flow of the key update routine to compute the succeeding sub-key \(K^{i+1}\) of round \((i + 1)\) from a given 128-bit sub-key \(K^i\) of round \(i\). Since the operations for loading or storing a byte and Galois Field (GF) additions each require one clock cycle, the entire key schedule routine takes 49 clock cycles to update sub-keys.

**Round Function**

In the following, we describe the integration and realization of the atomic AES operations within our developed encryption architecture. Benefiting from previously described design considerations, we could construct a hardware architecture using a minimal number of resources and slices of a selected Spartan-6 device which requires 97 clock cycles per round.

**ShiftRows:** This operation can be realized by using address translation for either the loading or storing addresses of the state register. For our design, we decided to change the storing address according to the ShiftRows function. Note, that this is also the reason for inherent shuffling support, since it allows to easily randomize the execution order of the bytes only by adapting the address translation function using a random permutation.

**SubBytes:** Trying to optimize and minimize the S-box as far as possible, we could identify the look-up table approach to result into the most compact S-box representation for FPGAs using 6-input LUT technology. Still, the S-box requires 8 slices, whereby each slice realizes a single 8-to-1 Boolean function using internal multiplexers (requires Slice-L or Slice-M).

**MixColumns:** When breaking down the MixColumns operation to its basic functions, it can be observed that only few different operations have to be implemented in order to support the entire operation: multiplications with small constants (i.e., 0x01, 0x02 and 0x03) as well as simple XOR addition. However, an implementation of the MixColumns operation is challenging, since the entire function is executed column-wise rather than byte-wise.
5.3 Implementation

In order to serialize the execution of MixColumns, we have to introduce an 8-bit wide register to hold intermediate values. This additional register along with preceding logic implements some accumulator-like logic (as shown in Figure 5.1) that provides four simple operations:

\[ r_i = x_i \]  \hspace{1cm} (write \ x \ into \ the \ register \ r)
\[ r_i = r_{i-1} + 01 \cdot x_i \]  \hspace{1cm} (add \ x \ multiplied \ by \ 01 \ to \ r)
\[ r_i = r_{i-1} + 02 \cdot x_i \]  \hspace{1cm} (add \ x \ multiplied \ by \ 02 \ to \ r)
\[ r_i = r_{i-1} + 03 \cdot x_i \]  \hspace{1cm} (add \ x \ multiplied \ by \ 03 \ to \ r)

Besides a 2-bit control signal to select the current operation, a small 2-bit counter is required to decide which column is processed currently in order to select and load the required bytes in the correct order.

AddRoundKey: Before a MixColumns operation can be started, the register of our MixColumns unit is initialized with the current sub-key byte in order to merge the key addition with the MixColumns operation. Then, the results from of Galois Field (GF) multiplications are added to the register iteratively so that it finally holds the accumulated result of both operations.

In addition, each round iteration uses a context-switching approach due to data dependencies during the execution. In particular, the MixColumns operation is critical in that sense since each result is calculated based on all four input bytes of a column. Moreover, since the bytes cannot be discarded before all four resulting bytes have been computed, we have to store the intermediate results at different locations in order to avoid overwriting of still required values. Fortunately, the LUTRAM holding the state is configured as a 32 × 8-bit memory and thus can hold up to two different 128-bit intermediate values. Hence, we can alternately read from one slot and store to the second slot without breaking data dependencies and overwriting still required values.

Last Round

After applying the SubBytes operation for the final round, the four additions and multiplications of the MixColumns operation are skipped. Due to the inherent latency of the additional intermediate register within the MixColumns unit, each correct ciphertext (after the final key addition operation) byte is returned after two clock cycles, hence the last round is computed in 33 clock cycles.

Control Unit

Decreasing the complexity of round logic usually comes at cost of increasing the complexity of control logic. In general, the controlling logic is in charge of managing the addressing of internal registers, input selection of multiplexers, control of the current MixColumns sub-operation and the state transitions of the internal Finite State Machine (FSM) in order to realize the entire AES encryption. Therefore, it implements several counters (round and current byte counter) which can be implemented in a single slice. Besides the internal FSM which has 32 different
states and requires five flip-flops to store and five LUTs to update the state. Most complex part of the controlling logic is the generation of internal control signals including the addressing of state registers. In particular the address generation is costly since it has to consider and perform the \textit{ShiftRows} operation. All remaining control signals are generated using a microcode-like approach, where the control signals are stored in ROM and are selected depending on the current state of the FSM.

### 5.3.2 Adding Side-Channel Protection

A common and basic way to hide leakage within measured power consumption is shuffling of operations. A random execution order makes it more difficult for an attacker to predict the internally processed data of the device and thus hampers side-channel analysis of the implementation. Fortunately, shuffling of the byte update process during the round computation can be implemented by randomizing internal addresses (as discussed before) and is supported inherently by our design. In the following section we first explain how to determine random permutations with modest area overhead and in a second step, we present the extension of our basic encryption architecture which implements the proposed shuffling countermeasure.
5.3 Implementation

Permutation Generation

A uniform permutation $\text{Perm}$ of any $n$-element sequence $\text{Seq}$ can be generated using a linear-time algorithm that applies $n$ swapping operations to $\text{Seq}$. The algorithm starts with $i = 0$ and iterates over all elements $s_i \in \text{Seq}$ with $i \leq n - 2$ and swaps the selected element with a random element of the remaining part of the sequence i.e., $\{s_{i+1}, \ldots, s_{n-1}\}$. However, sampling random numbers from $\{i, \ldots, n-1\}$ is not trivial and unfeasible for lightweight implementations, because it requires either modulo operations or algorithms with probabilistic run time. Hence, we decided to follow the approach of Veyrat-Charvillon et al. [VMKS12] who proposed to sample from $\{0, \ldots, n-1\}$. According to their investigation, this approach results in a slightly biased permutation, but they conclude that in the context of side-channel analysis this can be neglected. Following this approach, we store two initial permutations $\text{Perm}_0 = \{0, \ldots, 15\}$, $\text{Seq} = \{0, \ldots, 15\}$ in a $64 \times 4$-bit memory. In particular, $\text{Seq}$ is necessary to realize operations that cannot be shuffled (such as the key update function). The memory is succeeded by a 4-bit register which holds intermediate values during the swapping operation of the permutation algorithm that is interleaved completely with the phase of receiving and storing the initial plaintext and key bytes.

Shuffling of Round Update Order

In general, a new permutation $\text{Perm}_i$ is derived from the previous state $\text{Perm}_{i-1}$ stored in the memory using 64 bit of randomness. The permutation process is executed in parallel to the key-whitening of each encryption. During the remaining steps of the encryption process, the address signals are reconnected to the permutation memory instead of to the state registers. Then, the permutation generator serves as the translation unit which changes the address signals according to the current permutation $\text{Perm}_i$ and thus shuffles the byte update process after the MixColumns operation of each round. Otherwise, the second sequence $\text{Seq}$ is applied in order to avoid the shuffling, e.g., in case the key schedule is executed which requires a fix execution order due to data dependencies.

5.3.3 Pseudo Random Number Generation

Since symmetric block ciphers provide good statistical properties and are easy and fast to evaluate, they allow generating larger amounts of randomness starting from a relatively small seed. In order to design an AES implementation as lightweight as possible which still provides basic protection against side-channel attacks, sufficient amounts of randomness are required. Hence, an obvious approach is to reuse our AES core itself in order to extract the required randomness from an external random seed and build a self-contained side-channel protected lightweight AES instance. Since 64-bit fresh randomness are required for every encryption while the AES core can provide 128-bit at once, a small 2-bit counter controls switching to RNG mode after two consecutive encryptions. During the operation as RNG, the AES core returns a busy signal and does not accept encryptions.
Table 5.1: Comparison of selected implementations of AES for FPGAs.

<table>
<thead>
<tr>
<th>Design/Variant</th>
<th>Implementation</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data Path</td>
<td>Logic</td>
</tr>
<tr>
<td></td>
<td>(Bit)</td>
<td>(LUT)</td>
</tr>
<tr>
<td>Virtex-5</td>
<td>[BSQ+08](^1)</td>
<td>128</td>
</tr>
<tr>
<td>Spartan-2</td>
<td>[GB05](^2)</td>
<td>8</td>
</tr>
<tr>
<td>Spartan-2</td>
<td>[GB05](^2)</td>
<td>8</td>
</tr>
<tr>
<td>Spartan-3</td>
<td>[CG03](^2)</td>
<td>32</td>
</tr>
<tr>
<td>Spartan-3</td>
<td>[CB12](^1)</td>
<td>8</td>
</tr>
<tr>
<td>Spartan-6</td>
<td>[RSQL04](^2)</td>
<td>32</td>
</tr>
<tr>
<td>This work (Basic)(^1)</td>
<td>8</td>
<td>84</td>
</tr>
<tr>
<td>This work (Shuffl.)(^1)</td>
<td>8</td>
<td>94</td>
</tr>
<tr>
<td>This work (PRNG)(^1)</td>
<td>8</td>
<td>112</td>
</tr>
</tbody>
</table>

\(^1\) encryption only, \(^2\) encryption and decryption

5.4 Results

Our basic encryption architecture is implemented in only 21 slices\(^1\) using 100% of the LUTs and 14% of available flip-flops. The design can operate at a maximum clock frequency of 105 MHz and requires 1471 clock cycles for a full encryption, i.e., the maximum throughput is 9.12 Mbps. Side-channel protection increases (due to the permutation generation unit) resource consumption by additional 10 LUTs and 6 flip-flops to a final size of 24 slices. Unfortunately, the critical path is affected, dropping the maximum clock frequency to 90 MHz and the throughput to 7.82 Mbps. Finally, our self-contained AES encryption core again increases the resource consumption by 18 LUTs and 7 flip-flops to a final size of 28 slices. Note, however, that this design still provides the shuffling countermeasure. The final clock frequency is 75 MHz but due to the interleaving of encryption and RNG mode, the throughput for encryptions is further reduced by 33% to 4.35 Mbps.

5.4.1 Comparison

In Table 5.1 we list previous results of AES implementations for FPGAs reported in the literature. Note, however, that many of the listed implementations were designed for older Spartan-3 devices that do not provide comparable features, such as 6-input LUTs and 256-bit Distributed Memory and thus do not allow a fair comparison. Still, some related works have been implemented for both, Spartan-3 and Spartan-6 device families. In particular, recent work in [CB12] provides details for a Spartan-6 based design. The authors optimized the application of Shift Register instances and report final area numbers of 80 slices on an XC6SLX4 device. In addition, all results were obtained after place-and-route based on a Xilinx Spartan-6 XC6SLX4 using Xilinx ISE 14.7. The dense packing into a minimum number of slices was achieved by applying several constrains and placing almost every component by hand.
although implementing an 8-bit wide data path, the data path for the MixColumns operation is extended to 32-bit at cost of higher resource consumption. Still, the reported throughput of the design is much higher than our moderate performance due to the smaller number of clock cycles required for a single encryption.

5.5 Side-Channel Analysis

For our practical side-channel evaluations, we used another Spartan-6 XC6SLX75 embedded on a SAKURA-G board [GIS14]. All traces of the instantaneous power consumption have been measured by means of a LeCroy digital oscilloscope which monitored the voltage drop over a 1Ω resistor placed in the $V_{dd}$ path. Each trace has been recorded via the embedded amplifier of the SAKURA-G platform at a sampling rate of 500 MS/s. In order to simulate best-case scenarios for an attacker, the design is running at a low frequency of 3 MHz in order to reduce noise caused by overlapping power traces. We opted to perform a key-recovery attack on the S-box input of the last round. However, since our design is self-contained in terms of random number generation, we only provided random plaintexts and observed corresponding ciphertexts while our design was protected by means of shuffling.

5.5.1 Evaluation under Lab Conditions

In a first step, our design is evaluated in an isolated environment assuming optimal conditions for any side-channel observer. In particular, we performed two different measurements: first, we evaluated the side-channel resistance of our unprotected design (without shuffling the byte
Chapter 5 Side-Channel Protected Low-Area AES Implementation

Figure 5.5: Side-channel evaluation results for a CPA using the Hamming weight power model (for S-box input of last round without shuffling the byte execution order) with arithmetic noise.

Figure 5.6: Side-channel evaluation results for a CPA using the Hamming weight power model (for S-box input of last round with shuffling of the byte execution order) with arithmetic noise.

execution order) and second, we examined the resistance and gain in security when activating our integrated side-channel countermeasure.

**Unprotected**

For verification of the setup and implementation, an initial measurement (without applying shuffling to the byte execution order) is conducted. In order to recover the secret key, we performed a CPA and used the Hamming weight power model as distinguisher. The results for attacking the S-box input of the last round guessing the fifth key-byte are exemplary shown in Figure 5.3. For this attack we measured up to 100,000 power traces, but the correct key byte could already be revealed after about 200 traces.

**With Shuffling Countermeasure**

After checking the setup and implementation for correctness, our shuffling countermeasure is applied during operation for a second measurement. Figure 5.4 provides the result of the CPA (using the same setting as for the preceding evaluation). This time, we measured 500,000 power traces for random inputs but fixed key. It is obvious that shuffling clearly improves the resistance of the design against side-channel attacks, in particular the S-box computation is randomly performed at different time instances. However, perfect resistance still cannot be achieved but a successful key-recovery can be impeded when additional burden (and countermeasures for the attacker) are present. Without this, a key-recovery is possible requiring about 50,000 power traces before the correct key hypotheses could be clearly distinguished.
5.5.2 Evaluation for Embedded Applications

In order to simulate a more realistic environment in which our AES-128 encryption core is embedded, we implemented an additional core that is performing arithmetic computations in parallel. This core occupied around 9,000 slices (80% of our target device). Again, we performed two different measurements in order to evaluate the impact of the side-channel countermeasure.

Unprotected

Obviously, when increasing the arithmetic noise of the measurements, the SNR decreases which can be seen in the reduced correlation coefficients. A successful key-recovery attack is possible, as shown in Figure 5.5, but now requires around 5,000 power traces.

With Shuffling Countermeasure

Eventually, we again implemented the shuffling of the byte execution order while this time, in contrast to the side-channel evaluation under lab conditions, we simulated arithmetic noise using a second core operated in parallel. The evaluation results of this measurement can be seen in Figure 5.6. For a successful and unambiguous key-recovery, at least about 800,000 power traces have to be recorded.

Our results show that the spending of 7 additional slices for countermeasures can improve the resistance to physical attacks such as DPA or CPA significantly, in particular under practical conditions considering background noise introduced by additional cores running in parallel. Lightweight characteristics and properties of our design are still preserved although the encryption core is reused for random number generation. But even if side-channel protection is not mandatory, our architecture can still be used to generate pseudo randomness which then can be used for other purposes.

5.6 Conclusion

In this chapter we proposed three extremely lightweight architectures of an AES-128 encryption for recent Xilinx FPGAs. Our designs extensively exploit the 256-bit Distributed Memory provided within a single slice of modern Xilinx FPGAs as well as the availability of 6-input LUTs. The presented architectures provide throughputs between 9.12 and 4.35 Mbps which are sufficient for many applications with moderate requirements on performance but strict constraints on the resource consumption for cryptographic instances. Furthermore, we showed that side-channel protection and randomness generation can be included in the same core while maintaining lightweight properties of the implementations required by many security applications.
Chapter 6
Exploring RFC 7748: Curve25519 with Side-Channel Protection

In this chapter, we introduce an efficient architecture of the elliptic curve Curve25519 optimized for modern Xilinx 7-series FPGAs which performs up to 2519 point multiplications per second at a moderate resource cost of 1029 logic slices and 20 DSPs. Moreover, this basic design can be extended easily to a multi-core system for high-performance applications and scarcely forfeits performance even in the presence of advanced countermeasures to thwart SCA. This chapter is mainly based on a joint work with Tim Güneysu that has been published in [SG14]. However, additional details on the side-channel protection and countermeasures have been published in [SG15].

Contents of this Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>53</td>
</tr>
<tr>
<td>6.2</td>
<td>Specification of Curve25519</td>
<td>55</td>
</tr>
<tr>
<td>6.3</td>
<td>Design Considerations</td>
<td>56</td>
</tr>
<tr>
<td>6.4</td>
<td>Implementation</td>
<td>58</td>
</tr>
<tr>
<td>6.5</td>
<td>Results</td>
<td>64</td>
</tr>
<tr>
<td>6.6</td>
<td>Evaluation</td>
<td>67</td>
</tr>
<tr>
<td>6.7</td>
<td>Conclusions</td>
<td>71</td>
</tr>
</tbody>
</table>

6.1 Introduction

Efficient key agreement, exchange, and digital signatures are among the most crucial problems that cannot be solved solely relying on symmetric cryptographic primitives but require asymmetric (public-key) systems. For security-critical embedded applications, ECC has become the predominant asymmetric cryptographic system although it still involves complex modular arithmetic that is a particular burden for small embedded systems and processors. However, recent revelations on manipulations and back-doors have undermined the confidence in existing schemes and led to discussion among researchers and standardization bodies on the selection of new curves and the rigidity of existing curve generation processes. Within these discussions, the Internet Engineering Task Force (IETF) TLS working group requested new recommendations on elliptic curves for the next generation of TLS on which the Internet Research Task
Chapter 6 Exploring RFC 7748: Curve25519 with Side-Channel Protection

Force (IRTF) Crypto Forum Research Group (CFRG) has selected two candidates for the RFC 7748: Curve25519 and Curve448. Simultaneously, the IETF curlde group proposed both curves for application in future protocols such as DNSSEC. With further emergence of the IoT, TLS (including Curve25519 and Curve448) has to be implemented on various embedded and constrained devices. But since both curves were primarily designed for powerful software platforms (x86) and with hardly any evaluation of their resistance against low-level physical threats such as side-channel attacks, we still need to investigate the hardware implementation of both curves in combination with according countermeasures.

6.1.1 Related Work

Since there is a wealth of publications addressing ECC hardware architectures and side-channel countermeasures for ECC implementations, we refer an interested reader to the overviews in [dDQ07, FGM+10, FV12] and restrict the discussion of related work to the most relevant ones.

As one of the first attempts in hardware-based ECC implementations, Orlando and Paar [OP01] proposed a design targeting explicitly reconfigurable hardware using Montgomery-based multiplications including a series of precomputations. This publication was followed by many more, e.g., [GP08] trying to improve performance on FPGAs by using of dedicated multipliers or [SMB+06] trying to improve the performance by an algorithmic approach. Using integrated DSPs both for modular addition and multiplication was initially proposed in [°BPV03] targeting the standardized NIST primes P-224 and P-256 using a special reduction scheme.

However, efficient computation on constrained embedded devices is only one side of the coin, but still achieving physical security against side-channel analysis is the other. One of the first DPA attacks on ECC hardware implementations was presented in [M°PV07]. To prevent common DPA attacks, Coron proposed and implemented a set of countermeasures, including scalar and projective coordinate randomization, for a design performing Diffie-Hellman and ElGamal operations on elliptic curves [Cor99].

6.1.2 Contribution

In this chapter, we present a side-channel protected, efficient, and high-performance architecture of a special ECC system implementing the particular elliptic curve instance Curve25519 [Ber06] on reconfigurable hardware. In general, the main target application of this work is a single point multiplication operation suitable for high-performance applications, i.a. necessary for Diffie-Hellman key agreement protocols and digital signature generation and verification. Particularly, we extend our initial work presented in [SG14] to provide high-performance cryptography including physical protection against common side-channel analysis such as timing behavior, SPA and DPA. Our design takes particular advantage of the arithmetic cores intended for digital signal processing provided by most modern reconfigurable devices. Although Curve25519 was initially proposed to accelerate the Diffie-Hellman key agreement primarily in software, we show that its characteristics can be similarly exploited in hardware to achieve and implement a compact and high-performance ECC processor on modern reconfigurable devices. Moreover, our multi-core Curve25519 architecture, implemented on a moderate Xilinx XC7Z020 FPGA, achieves more than 32 000 point multiplications per second and still more than 27 500 operations when enhanced with DPA countermeasures. Providing these results, our de-
sign can virtually support any high-performance application of asymmetric cryptography, even with high demands on the physical security of the implementation.

### 6.2 Specification of Curve25519

Curve25519 is an efficient elliptic curve (more precisely a Montgomery curve) specified over a prime field with a prime of shape $2^n - c$, i.e., close to a power of two (Pseudo Mersenne prime), and defined by its short Weierstrass equation:

$$E_M : y^2 = x^3 + 486662x^2 + x \mod (2^{255} - 19) \quad (6.1)$$

In general, according to RFC 7748, Curve25519 provides the function $X_{25519}$ in order to accelerate the Diffie-Hellman key agreement, but also can be used for digital signature schemes based on its birationally equivalent Edwards curve Ed25519. However, the function $X_{25519}$ defines a point multiplication scheme that provides inherent resistance against analysis of the timing behavior and a conjectured security level comparable to NIST P-256 or AES-128. Based on the point multiplication operation, two parties eventually can derive a 255-bit shared secret according to the Diffie-Hellman key agreement protocol involving their public and private keys.

#### 6.2.1 Field Arithmetic

On the lowest level, Curve25519 processes 255-bit values and performs arithmetic operations over $GF(p)$ defined by $p = 2^{255} - 19$. In general, field arithmetic operations include modular addition, subtraction, multiplication, squaring, and inversion. Since the underlying finite field is defined over a Pseudo Mersenne prime, providing a special structure, efficient reduction can be performed using the fact that $2^{255} \equiv 19 \mod p$. Eventually, inversion can be reduced to modular multiplication using Fermat’s Little Theorem (FLT), i.e., $a^{p-2} \equiv a^{-1}$. Hence, given a set of modular addition, subtraction and multiplication operations and instructions, we can realize any functionality that is necessary in order to implement group arithmetic operations as well as the point multiplication.

#### 6.2.2 Group Arithmetic

According to RFC 7748, the function $X_{25519}$ defines a variable scalar-point multiplication on the Montgomery curve (Curve25519) using the Montgomery ladder algorithm [Mon87] that allows performing a point addition and a point doubling in a single, combined step. Hence, given $d = 121665$, two points $Q_1, Q_2 \in \mathcal{E}$ (in projective coordinate representation) and the difference $Q_3 = Q_1 - Q_2$, a single step of the Montgomery ladder algorithm computes two points $Q_4, Q_5 \in \mathcal{E}$ such that $Q_4 = 2 \cdot Q_1 = (x_4, z_4)$ is the result of the point doubling with:

$$x_4 = (x_1 - z_1)^2 \cdot (x_1 + z_1)^2 \quad (6.2)$$

$$z_4 = 4x_1z_1 \cdot (x_1^2 + dx_1z_1 + z_1^2) \quad (6.3)$$

and $Q_5 = Q_1 + Q_2 = (x_5, z_5)$ is the result of the point addition with:

$$x_5 = z_3((x_1 - z_1) \cdot (x_2 + z_2) + (x_1 + z_1) \cdot (x_2 - z_2))^2 \quad (6.4)$$

$$z_5 = x_3((x_1 - z_1) \cdot (x_2 + z_2) - (x_1 + z_1) \cdot (x_2 - z_2))^2 \quad (6.5)$$
Fortunately, these equations are independent of the \( y \)-coordinate which can be omitted during the intermediate computation and only has to be restored for the final result using Equation 6.1. Figure 6.1 shows the algorithmic flow of a single step for three points \( Q_1, Q_2 \) and \( Q_3 \). Note, that every addition or subtraction is followed by a multiplication which again nearly always is succeeded by an addition or subtraction. This observation is very helpful for designing an efficient hardware architecture. In total, a single step of the Montgomery ladder algorithm involves 6 multiplications, 4 squarings, 4 additions, and 4 subtractions over \( GF(p) \). The number of operations performed and its sequence is always the same, independently of the processed data. Therefore, the computation can be executed in constant time preventing timing-based attacks and even SPA.

### 6.2.3 Point Multiplication

For variable scalar, but fixed base-point multiplication \( k \times P \) on Curve25519, a total of 255 combined point doubling and addition operations (single steps of the Montgomery ladder algorithm) are executed followed by a final inversion of \( z \) and a single field multiplication \( x \times z^{-1} \) computing the final result in its affine coordinate representation. Depending on the application, the resulting \( x \) value can be directly used in a cryptographic protocol (e.g., to derive a shared secret using an additional call to a hash function), or the missing \( y \)-coordinate can be restored using Equation 6.1. Moreover, since the point multiplication directly depends on the secret scalar \( k \), particular care has to be taken in order to prevent side-channel leakage of secret information.

### 6.3 Design Considerations

In this section, we discuss some key features of Curve25519 that open attractive possibilities to efficiently map its mathematical and arithmetic structures to modern reconfigurable hardware devices.
Prime Field. For most modern standardized elliptic curves considered for high-performance applications, e.g., the NIST P-224 and P-256 instances, the underlying prime field is based on a Generalized Mersenne Prime which allows a modular reduction solely based on additions and subtractions. However, the underlying finite field of Curve25519 is based on a Pseudo Mersenne Prime of shape $2^n - c$ providing a slightly different modular reduction scheme. In particular, for the elliptic curve instance Curve25519, the reduction can be computed by a multiplication with the small constant $c = 19$ and additional additions or subtractions.

Modular Multiplication. Any field element, with a total size of 255 bits, can be divided perfectly into fifteen chunks, each having a size of 17 bits. Processing these smaller words is usually inefficient for common processors and microcontrollers which often operate on a basis of 8-, 16-, 32- or 64-bit data words. Modern FPGA devices, however, provide a multitude of dedicated, full-custom designed, arithmetic DSP slices equipped with preaddition, multiplication and accumulation or addition stages for fast integer arithmetic. Each DSP instance is enhanced with additional register stages to reduce the internal critical path delay and allow operations at maximum device frequency. Since the multiplication unit within each DSP block was designed to support signed $25 \times 18$-bit wide multiplications (or $24 \times 17$-bit unsigned operations), this is a perfect fit to our requirement of processing unsigned 17-bit data words. By means of a customized, interleaved multiplication scheme, multiplying two 255-bit field elements can be rearranged and distributed over several DSP blocks operating in parallel. Each single DSP unit then has to compute only one $17 \times 17$ bit multiplication at a time resulting. Additionally, we can use the included accumulation stage within each DSP core to sum up the intermediate results into a partial product. Eventually, the full multiplication can be performed using 15 DSP slices in parallel and we end up generating the final result in the accumulation stage that is slightly too large but which can be reduced in a subsequent recombination step. The final reduction step itself can be implemented by a constant multiplier with $c = 19$, realignment logic to combine shares of partial products as well as a subtraction stage to correct the result by reducing it modulo $P$ in case it is still slightly too large.

Modular Addition and Subtraction. Besides modular multiplication, the basic group operations (point doubling and addition) require the computation of modular additions and subtractions. Fortunately, the addition respectively subtraction unit can be implemented as simple cascade of only two DSP instances, one to perform the main arithmetic operation and one for the subsequent modular reduction.

Modular Inversion. In order to return a unique result for each point multiplication operation, a final inversion is required to convert the internal result based on projective coordinates back to an affine coordinate representation. For the basic version of a modular inversion, we make use of FLT. This approach requires solely the modular multiplier already provided by the core that is augmented with a small additional state machine. Inversion based on this approach requires negligible extra resources, however, its performance will be comparably slow. Another approach would be to implement a dedicated inversion unit based on the binary Extended Euclidean Algorithm (EEA). This extra circuit requires a significant amount of additional resources but the computation of each inversion is signifi-
Chapter 6 Exploring RFC 7748: Curve25519 with Side-Channel Protection

Figure 6.2: Overview of the Curve25519 core

- Significantly faster. Still, in a multi-core scenario, these hardware costs can be mitigated mostly by sharing one dedicated inversion unit among several point multiplication cores.

Memory. Eventually, we limit the storage requirements to a bare minimum, using only two 36K dual-port BRAM units to store all intermediate values in a butterfly-wise configuration and data flow using an enhanced 34-bit data path to process two words at the same time.

6.4 Implementation

In this section, we first present details of a single-core Curve25519 architecture resulting in moderate performance at low resource costs. Second, we extend our design into a multi-core architecture that aims to provide maximum throughput on a chosen XC7Z020 FPGA.

6.4.1 Single-Core Architecture

Our single-core Curve25519 implementation is designed as a supplementary unit to provide asymmetric, public-key cryptographic operations, but to save most of the FPGA logic and resources for additional main applications. In detail, the cryptographic core is designed to support point multiplications using points on Curve25519. Internally, the single-core architecture supports addition and doubling operations of points in projective coordinate representations. However, for use with most cryptographic protocols, the core additionally implements an inversion functionality to convert final results from projective back to affine coordinate representations. Internally, the arithmetic processor therefore supports two basic operation modes: either a combined double-and-add step function using the Montgomery ladder algorithm or a single modular multiplication operation. In particular, the latter instruction is required for the final conversion, i.e., for a modular inversion of a field element \( a \in GF(p) \) by computing \( a^{p-2} \mod p \) based on FLT. To prevent timing attacks, the arithmetic unit performs the point multiplication running a total 255 double-and-add operations and 266 iterated multiplications for the inversion, both in constant time (but so far without considering further, advanced side-channel analysis).
In general, our implementation follows several of the design suggestions of Bernstein for software implementations as given in the original work on Diffie-Hellman computations over Curve25519 [Ber06]. In particular, each addition or subtraction is followed by a subsequent multiplication again nearly always succeeded by another addition or subtraction. This fact leads to the design presented in Figure 6.2 using two dual-port BRAM units in a butterfly-like configuration. More precisely, the first BRAM only receives results of the addition or subtraction unit and provides inputs to the multiplication unit. Further on, the second BRAM stores all results of the multiplication core and feeds the addition unit. This way parallel operation of both arithmetic units is enabled and pipeline stalls through loading and write-back can be avoided with only little overhead.

**Modular Addition and Subtraction Unit**

Centerpiece of the modular addition and subtraction unit (computing \( c = a \pm b \mod p \)) are two DSP blocks supporting \( 25 \times 18 \)-bit signed multiplications and up to 48-bit additions, subtractions or accumulations. Whereas the first DSP always performs the main operation (i.e., addition or subtraction \( c' = a \pm b \)), the second DSP block computes a prediction for the reduced result by \( c'' = c' \mp p \). Both, \( c' \) and \( c'' \) are stored into the first BRAM and distinguished by a flag which is obtained from the last carry or borrow bit in the prediction operation and indicates at which address the correct result is stored. In total, modular addition or subtraction takes 10 clock cycles which can be executed in parallel to any multiplication operation. Thus, when exploiting the alternating, butterfly-like data flow as mentioned above, the latency for modular addition or subtraction is completely absorbed within the latency for a concurrently running modular multiplication.
6.4.2 Multi-Core Architecture

A main caveat of the single-core architecture is the slow inversion. In this section we augment the previously described architecture with a dedicated inversion circuit and share it among several point multiplication cores for an optimal cost-performance trade-off. The number of core instances per inversion unit is upper-bounded by the available resources on the respective device as well as the latency of each point multiplication with respect to one final inversion. Since these numbers directly correspond to resources available on a given FPGA, we implemented the design generically to provide maximum flexibility and facilitate scaling to any other device. Figure 6.4 depicts the new communication interface and the additional controller for distributing incoming data packets among the Curve25519 cores. Unlike our single-core introduced in Section 6.4.1, all cores of this architecture only support the elementary step function (double-and-add operation) but no modular inversion since this computation finally is performed in a subsequent step by a dedicated unit shared by all cores.
6.4 Implementation

Multi-Core Controller

Core 1
Core 2
Core 11

(a) First core is loaded with data, marked busy and starts computing.

Multi-Core Controller

Core 1
Core 2
Core 11

(b) Next data is handed over to Core 2 which is marked busy and starts operating, too.

Multi-Core Controller

Core 1
Core 2
Core 11

(c) As long as data is available and cores are not busy, a new core is triggered with data.

Multi-Core Controller

Core 1
Core 2
Core 11

(d) After finishing calculation, Core 1 hands over its result to the inversion unit and is marked ready again.

Figure 6.5: Round-Robin-based load balancing scheme

Dedicated Inversion Unit

In many cases the division in finite fields is the most expensive operation involving a modular inversion and at least one multiplication. In our single-core approach, we noticed that inversion based on FLT is rather costly since the computation requires roughly 20% of the run time of an entire CURVE25519 point multiplication. Therefore, we opted to implement a modular inversion circuit based on the binary EEA as an extension to the existing core. Our inversion unit uses wide addition and subtraction units and well-known implementation techniques so that we refrain from giving all details on the implementation. With respect to the multi-core design approach, the inversion receives both $x$ and $z$ as inputs, and computes the final result $\frac{x}{z}$ in constant time. Since this kind of inversion is significantly faster compared to a single point multiplication operation, the inversion core can serve as subsequent, supplementary unit for several point multiplication cores (cf. Section 6.5).

Load Balancing

Due to the concurrent operation of individual cores, we implemented a scheme to distribute incoming data to any core that becomes available. This approach is basically a round-robin-like load balancing scheme where the controller unit keeps track of the last active and the next available core (cf. Figure 6.5). Loading continues until all cores are busy. As soon as one core reports a result and the inversion unit is idle, it is handed over to the inversion unit and the core is marked ready again and can be loaded with a next parameter set.

Clock Domain Separation

The point multiplication cores can operate at a clock frequency of 200 MHz. However, since the dedicated inversion unit implements 256-bit wide additions and subtractions in fabric logic it only supports a maximum clock frequency of roughly 130 MHz. Still, in order to operate the implementation at maximum speed, we use different clock domains for the point multiplication cores on the one hand (200 MHz) and the controller and inversion unit on the other hand (100 MHz).
Chapter 6 Exploring RFC 7748: Curve25519 with Side-Channel Protection

6.4.3 Side-Channel Countermeasures

In [Cor99], several countermeasures were suggested to counteract DPA attacks on elliptic curve hardware implementations. Since our design is inherently resistant against timing (cf. Section 4.3.1) and SPA attacks (cf. Section 4.4.1), we now present an extension of the Curve25519 core that adopts the proposed countermeasures against DPA attacks. In the following section we will briefly explain the idea of side-channel countermeasures and describe their smooth integration into the core architecture.

Source of Randomness

In Section 4.5, we mentioned two major approaches to counter DPA: randomizing (i.e., masking and blinding) or hiding (i.e., reduction of the SNR) side-channel leakage within the power consumption. Since power equalization schemes (in order to hide the leakage) are a hard problem, especially for FPGAs with their fixed routing and logic structure, we will focus our countermeasure approaches according to randomization techniques. Therefore, our countermeasures require an additional RNG delivering the core with at least 285 bits of fresh randomness during every operation (255 bits for the randomized projective coordinates and the random $\lambda$-coordinate $\lambda$, 24 bits for the scalar blinding factor $r$ and 6 bits for the seed of address randomization Linear-Feedback Shift Register (LFSR)). Since the focus of this work is the implementation and integration of side-channel countermeasures into our Curve25519 architecture and not the design of RNGs, we assume that the randomness is provided by an external True or Pseudo Random Number Generator (PRNG), e.g., based on a stream or block cipher.

Scalar Blinding

Most SPA and DPA attacks on elliptic curve implementations aim to reveal the secret scalar applied during the point multiplication operation. In order to thwart these kinds of attacks designers need to decorrelate the signatures of computations that involve the secret scalar from the actual power consumption of a physical device. Since the point multiplication in ECC depends on two parameters, a public point $P$ and a secret scalar $k$, both can be target for randomization. In our first approach, the private scalar is randomized and blinded by adding random multiples of the group order of the underlying elliptic curve. More precisely, using a scalar $r$ corresponding to the group order $\#E$ of the elliptic curve, the point multiplication $r \times P$ always results in the point at infinity $O$ which is the neutral element for the point addition. Hence, if we choose a scalar $k'$ in a way such that

$$k' = k + r \times \#E,$$

then it holds for the point multiplication of the new scalar $k'$ and a point $P$ that

$$k' \times P = (k + r \times \#E) \times P = k \times P + r \times O = k \times P.$$

In [Cor99], a bit size of at least 20 bits is suggested as sufficiently secure for the random value $r$.\(^1\) By using a single DSP block for accelerating computation of $k'$, we can even pick up to 24

---

\(^1\)In fact, due to the special structure of the group order $\#E$ it turns out that 24-bit blinding factors are insufficient to hide the structure which could be exploited using advanced SCA [SW15]. Thus, in order to prevent these kind of attacks, practical solutions have to rely on blinding factors of at least half of the bit size of the group order, i.e., about 128 bits or more should be considered for Curve25519.
random bits for \( r \) to serially compute the multiplication and final addition with the original \( k \) within 17 clock cycles. The final result \( \bar{k} \) has a maximum bit size of 280 bits, finally leading to 25 additional step function calls. Note, however, that the entire computation remains still constant in time.

**Randomized Projective Coordinates**

Besides predicting parts of the random scalar, an attacker could try to guess parts of the binary representation of elliptic points and some intermediate values of the double-and-add formula in order to deduce the used scalar (even if its randomized). Internally, the affine coordinates \((x, y)\) of elliptic points are replaced by projective coordinates \((x, y, z)\) during computation whereby the computations for Curve25519 only depends on the coordinates \((x, z)\) and initially assumes \( z = 1 \). This representation is usually used to relax the computation and to replace costly inversions of the point multiplication by additional field multiplications. To protect against attacks trying to guess points on the elliptic curve, [Cor99] proposes a randomization of the projective representation which can be applied easily for our design. In order to randomize the point without affecting the result, we choose to multiply all coordinates with a random 255-bit \( \lambda \) during the initialization phase of the core. We therefore use

\[
\mathcal{R} = (x_\mathcal{R}, z_\mathcal{R}) = (\lambda x_p, \lambda z_p) = (\lambda x_p, \lambda),
\]

obtained from multiplication in our core so that the original initial point \( \mathcal{P} = (x_p, z_p) = (x_p, 1) \) stored in the BRAM is replaced with a new, randomized point \( \mathcal{R} = (\lambda x_p, \lambda) \). Note, that all other values and constants of the computation are unaffected. The value \( \lambda \) is randomly chosen for each execution so that, even when always using the same scalar and curve point as input to the computation, all intermediate results differ during execution of the point multiplication but still lead to the correct result.

**Memory Address Scrambling**

For our implementation, all inputs of the Montgomery ladder algorithm \((Q_1, Q_2, \text{ and } Q_3)\) are stored in the same BRAM and after each execution step both results \( Q_4 = 2 \cdot Q_1 \) and \( Q_5 = Q_1 + Q_2 \) will overwrite \( Q_1 \) and \( Q_2 \) while serving as a new input to the next iteration of the algorithm execution. For every step of the double-and-add computation, a single bit of the secret scalar is evaluated and indicates whether the inputs have to be swapped or not. Thus,
the BRAM access pattern at the beginning of every execution of the double-and-add formula is key-dependent and can help an attacker to reveal information of the secret scalar. Therefore, an adversary who is able to recover information on the BRAM addressing pattern using DPA, can easily identify all inputs of the algorithm and thereby recover each bit of the secret scalar successively.

Hence, in order to protect the memory accesses against side-channel attacks and hide revealing access patterns, the addresses should be randomized for each execution. In case of our design, we decided to choose random addresses for the intermediate values and all results that are stored in the memory prior to each point multiplication call. After every execution, we then choose a new set of random addresses for the intermediate values and results that are stored in the BRAMs. Since all addresses of our BRAM have a size of 6 bits, we can select up to $2^6$ different sets of random addresses. With the help of a LFSR (based on the formula $x^6 + x^5 + 1$), we can determine all addresses depending on a random base address. Therefore, we take 6 bits, supplied by an external RNG, and use them as seed (i.e., the base address) for the LFSR. Before our core can perform a new point multiplication operation, the LFSR is advanced and the random addresses are saved for the next point multiplication, initializing and preparing the BRAM by rearranging all required values and constants. Hence, in total, we allow $2^6$ different sets of random addresses instead of a single fixed one. Besides other technical issues such as noise reduction, an attacker needs to distinguish between these memory lines. Since this classification problem is highly complex to solve when other countermeasure are present and running concurrently, we do not expect DPA attacks to be successful from a practical point of view.

6.5 Results

In this section we discuss performance and implementation results of both, our single-core and our multi-core design. All results were obtained after place-and-route based on a Xilinx XC7Z020CLG484-3 using Xilinx ISE 14.7.

6.5.1 Comparison

In Table 6.2 we provide the resource consumption for both, the basic and countermeasure enhanced version for our single-core and multi-core design, respectively.
### Table 6.2: Summary of the device utilization and performance.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Unprotected</th>
<th>Protected</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resource Utilization</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Look-Up Tables</td>
<td>2 783 (5.23%)</td>
<td>2 862 (5.38%)</td>
</tr>
<tr>
<td>Flip-Flops</td>
<td>3 592 (3.38%)</td>
<td>3 784 (3.56%)</td>
</tr>
<tr>
<td>Logic Slices</td>
<td>1 029 (7.73%)</td>
<td>1 180 (8.87%)</td>
</tr>
<tr>
<td>Digital Signal Processors</td>
<td>20 (9.09%)</td>
<td>22 (10.00%)</td>
</tr>
<tr>
<td>Block-RAMs</td>
<td>2 (1.43%)</td>
<td>2 (1.43%)</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step Function (200 MHz)</td>
<td>323.85 µs (64770 cy.)</td>
<td>344.40 µs (68880 cy.)</td>
</tr>
<tr>
<td>Inversion (200 MHz)</td>
<td>73.15 µs (14630 cy.)</td>
<td>71.86 µs (14372 cy.)</td>
</tr>
<tr>
<td>Point Multiplication (200 MHz)</td>
<td>397.00 µs (79400 cy.)</td>
<td>416.26 µs (83252 cy.)</td>
</tr>
</tbody>
</table>

| **Resource Utilization**   |             |           |
| Look-Up Tables             | 34 009 (63.93%) | 30 582 (57.48%) |
| Flip-Flops                 | 43 875 (41.24%) | 39 916 (37.52%) |
| Logic Slices               | 11 277 (84.79%) | 10 777 (81.03%) |
| Digital Signal Processors  | 220 (100.00%)  | 220 (100.00%)  |
| Block-RAMs                 | 22 (15.71%)    | 20 (14.29%)    |
| **Performance**            |             |           |
| Step Function (200 MHz)    | 323.85 µs (64 770 cy.) | 344.40 µs (68 880 cy.) |
| Inversion (100 MHz)        | 16.67 µs (1 667 cy.)  | 16.67 µs (1 667 cy.)  |
| Point Multiplication (100 MHz) | 340.52 µs (34 052 cy.) | 361.07 µs (36 107 cy.) |

### Single-Core Architecture

The single-core implementation requires only a moderate amount resources on the Xilinx XC7Z020 device, i.e., 7% (8%) of the LSs and 9% (10%) of provided DSP blocks for the (protected) design. The remaining device components are available for any other function or application that are required to be implemented. Both single-core designs, i.e., the protected and unprotected solution, have a maximum operation frequency of 200 MHz. While the unprotected single-core needs 254 clock cycles to compute a single iteration of the step function and another 14 630 cycles for the final inversion, this could even be decreased for the protected alternative by a slightly optimized modular multiplication unit. A step function now requires only 246 and the final inversion only 14 372 clock cycles. But due to the implementation of additional countermeasures, some delay during the computation is added (cf. Table 6.1) which leads to about 83 000 for a total point multiplication instead of roughly 80 000. Note that still about 20% of the clock cycles are required to compute the final inversion. Besides, again due to the implementation of additional countermeasures, our protected single-core design has a slightly higher demand of resources, in particular on registers and DSP instances. In this regard, the most resource-consuming countermeasure is blinding of the secret scalar, as one can see in Table 6.1. Fortunately, based on further optimization of our single-core, the resource difference between the protected and unprotected alternative is much smaller than the overhead introduced by all countermeasures. In summary, both single-core architectures, protected and unprotected, can perform about 2500 point multiplications per second while utilizing less than 10% of the chosen Xilinx FPGA.
### Table 6.3: Selected high-performance implementations of ECC

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Device</th>
<th>Security</th>
<th>Architecture</th>
<th>Performance</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>LS</td>
<td>DSP</td>
<td>BRAM</td>
</tr>
<tr>
<td><strong>Curve25519</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single-Core⁴</td>
<td>XC7Z020</td>
<td>127 bit</td>
<td>20</td>
<td>2</td>
<td>79400</td>
</tr>
<tr>
<td>Multi-Core⁴</td>
<td>XC7Z020</td>
<td>127 bit</td>
<td>220</td>
<td>20</td>
<td>34052</td>
</tr>
<tr>
<td>Single-Core⁵</td>
<td>XC7Z020</td>
<td>127 bit</td>
<td>20</td>
<td>2</td>
<td>83252</td>
</tr>
<tr>
<td>Multi-Core⁵</td>
<td>XC7Z020</td>
<td>127 bit</td>
<td>220</td>
<td>22</td>
<td>36180</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Curve25519</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software</td>
<td>Intel Xeon E3</td>
<td>127 bit</td>
<td>—</td>
<td>—</td>
<td>156148</td>
</tr>
<tr>
<td>Software</td>
<td>AMD FX-8120</td>
<td>127 bit</td>
<td>—</td>
<td>—</td>
<td>193514</td>
</tr>
<tr>
<td>Software</td>
<td>Cortex-A9+NEON</td>
<td>127 bit</td>
<td>—</td>
<td>—</td>
<td>567267</td>
</tr>
<tr>
<td>Software</td>
<td>Snapdragon S3</td>
<td>127 bit</td>
<td>—</td>
<td>—</td>
<td>410128</td>
</tr>
<tr>
<td><strong>NIST P-256</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single-Core</td>
<td>XCC4VFX12</td>
<td>128 bit</td>
<td>32</td>
<td>11</td>
<td>240000</td>
</tr>
<tr>
<td>Multi-Core</td>
<td>XCC4VFX12</td>
<td>128 bit</td>
<td>512</td>
<td>176</td>
<td>-</td>
</tr>
<tr>
<td><strong>GF(p) (any)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single-Core</td>
<td>XC2VP125</td>
<td>128 bit</td>
<td>256</td>
<td>0</td>
<td>150000</td>
</tr>
<tr>
<td>Single-Core</td>
<td>XC4</td>
<td>128 bit</td>
<td>37</td>
<td>11</td>
<td>110000</td>
</tr>
<tr>
<td>Single-Core</td>
<td>XC5</td>
<td>128 bit</td>
<td>37</td>
<td>10</td>
<td>110000</td>
</tr>
</tbody>
</table>

¹ unprotected, ² protected

## Multi-Core Architecture

Our multi-core implementation is optimized for high performance still targeting a moderately-large FPGA and making use of all resources available. Due to the clock domain separation, the multi-core architecture can compute a point multiplication in about 34 000-36 000 cycles at a frequency of 100 MHz. In addition, up to 11 point multiplication operations (or 10 using cores enhanced with side-channel countermeasures) can be performed in parallel with an additional initial latency of 1667 clock cycles (which is the time that is required for the inversion). In case of the side-channel protection, we can use only 10 cores in parallel because the limiting resource is still the number of DSP slices available and our scalar blinding countermeasure requires two additional DSPs for a single core. Still, the multi-core approach leads to a final throughput of more than 32 000 (or 27,500 respectively) Curve25519 operations per second.

## 6.5.2 Comparison to Other Work

Despite the fact that this is, to the best of our knowledge, the first implementation of Curve25519 in hardware and that there exist only few implementations in software, we extend the scope for our comparison to implementations over a different type of elliptic curves. In Table 6.3 we list implementation results for various platforms that implement ECC on a comparable security level to Curve25519. Note, however, that due to varying technology of different FPGA generations a fair comparison is actually not accurately possible.

In [GP08] the standardized NIST P-256 curve was implemented using a similar approach on a Xilinx Virtex-4. The authors report their design to perform point requires more resources at a performance of about 2 000 point multiplications per second. A more recent result, published in [MLPJ13], implements a general ECC architecture over $GF(p)$ using DSP slices to enhance...
6.6 Evaluation

Eventually, we performed a side-channel vulnerability analysis and evaluation using practical power measurements obtained from a SAKURA-X [Lab] platform populated with a 7-series Kintex FPGA (XC7K160T). To this end, we measured the voltage drop over an 1 Ω resistor in the $V_{dd}$ path using a digital oscilloscope with a sampling rate of 250 MS/s, 20 MHz bandwidth limitation, and a stable, jitter-free 96 MHz clock.

6.6.1 Detection of Scalar-Dependent Leakage

During our analysis, we focused on detection of scalar-dependent leakage using a fixed base point (i.e., the default base point $x_P = 9$) and performed a non-specific $t$-test with fix and randomly chosen scalars according to [TG16]. To avoid any input-dependent static leakage, we applied two minor modifications to our original design. First, instead of receiving the base point in affine coordinate representation, the transformation to a projective coordinate representation (including the randomization) is already performed externally. Second, the scalar is provided in a blinded representation using a random and extended 128-bit blinding factor. Moreover, we excluded the final multiplication (after the inversion) from our measurements in order to avoid output-dependent leakage in our evaluation since this operation obviously results in an unmasked scalar-dependent value.

Further on, we defined four different evaluation profiles in order to gradually evaluate our main countermeasures (i.e., Random Projective Coordinates and Scalar Blinding). While for Profile 1, we disabled our PRNG in order to evaluate an unprotected implementation, Profile 2 and 3 investigate the security of our proposed countermeasures individually. Eventually, Profile 4 provides a combined analysis of both countermeasures and their effect on the side-channel security. In Figure 6.7, we provide two mean traces, both based on 1 000 measurements, where we at first disabled (Figure 6.7(a)) and then enabled (Figure 6.7(b)) all of our countermeasures.
Chapter 6 Exploring RFC 7748: Curve25519 with Side-Channel Protection

Profile 1: PRNG off

Our first profile serves as a reference setup where we disabled our PRNG in order to provide a constant zero value. Consequently, the secret scalar is provided in an unblinded fashion and the base point has a constant $z$-coordinate of $z_P = 1$. In Figure 6.8 we present the $t$-test based evaluation results after measuring 10,000 power traces for two different runs (with and without random BRAM addresses). Obviously, the $t$-test detects scalar-dependent leakage with very high confidence giving proof for the correctness of our setup. As expected, the leakage is mainly detectable during the last 255 Montgomery step executions and the final inversion, since the secret scalar is padded with zeros for both cases (fix and random scalar).

Profile 2: Randomized Projective Coordinates

The second evaluation profile investigates the individual security gain through the randomization of projective coordinate representations without scalar blinding countermeasures. Therefore, the core is provided with a random multiple of the $x = 9 \pmod{p}$ and the random $z$-coordinate. For this profile, we captured 100,000 power traces with fix vs. random scalars and the $t$-test results are shown in Figure 6.9 both with and without BRAM address scrambling. For both measurements, since the unprotected scalar is expanded by 128 bits and padded with leading zeros, we can clearly distinguish an initial phase that processes these zeros from the actual processing of the scalar bits due to its smaller $t$-values. In summary, randomizing projective coordinate representations helps to decrease the observable side-channel leakage though it does not prevent the implementation from exhibiting scalar-dependent side-channel information.

Figure 6.8: Non-specific $t$-test results with fixed base point: Profile 1 (10,000 traces)

Figure 6.9: Non-specific $t$-test results with fixed base point: Profile 2 (100,000 traces)
6.6 Evaluation

In Figure 6.10 we provide the evaluation results for our third profile in order to investigate our scalar blinding countermeasure. Again, we measured 100,000 power traces using a non-specific $t$-test setup with fix vs. random scalars. In addition, each scalar is blinded externally using a random 128-bit blinding factor (i.e., about half of the bit length of the group order) which is multiplied by the group order of the base point and added to the original scalar. Then, the core receives and processes only the blinded scalar. Obviously, this countermeasure avoids any detectable scalar-dependent leakage for the given number of measurements and in particular we cannot distinguish the loading and processing of any fix and random secret scalar (on the first statistical order).

Profile 4: Combination

Eventually, our forth profile combines all countermeasures in order to investigate their interaction. Figure 6.11 provides the non-specific $t$-test results using 1,000,000 power traces and as expected we cannot observe any first-order side-channel leakage for both with and without random memory addressing while using scalar blinding and randomized projective coordinates.

6.6.2 Detection of Base-Point-Dependent Leakage

In a further step, we evaluated the detection of base-point-dependent leakage, this time using a fixed scalar throughout the evaluation procedure while the base point is chosen from a random set and interleaved with operations on a fix base point. Again, we used $x_P = 9$ as fix base point. However, in order to ensure the correct operation of our core and the scalar blinding countermeasure, the random base point has to have the same group order than the default base point. To this end, we modified our measurement setup such that when a random base point is
sent to the core, an arbitrarily chosen scalar $s$ is used to derive a new base point as $P' = s \times P$, i.e., we performed an additional point multiplication in software to ensure the same group order for $P$ and $P'$. Besides, the scalar has been fixed to the test vector that is given in RFC 7748.

**Profile 1: PRNG off**

Again, the first profile is intended as a reference for our evaluation setup. To this end, we again disabled our PRNG and countermeasures. Consequently, the scalar is not blinded and the base point has a constant $z$-coordinate of $z_P = z_{P'} = 1$. Figure 6.12 presents the $t$-test results using 10 000 power traces (with and without memory address scrambling). As expected, the $t$-score exceeds the threshold significantly, i.e., we can detect base-point-dependent leakage with high confidence.

**Profile 2: Randomized Projective Coordinates**

In Figure 6.13 we provide the evaluation results when enabling point randomization as countermeasure. Obviously, we cannot detect any first-order side-channel leakage for our design after measuring 100 000 power traces.

**Profile 3: Scalar Blinding**

Figure 6.14 provides the evaluation results of our third profile, when enabling the scalar blinding countermeasure, using only 10 000 power traces and with fixed scalar. In contrast to Profile 3 with fixed base point, we can clearly detect and observe leakage for both cases. In particular the internal processing of the point $Q_3 = Q_1 - Q_2$ can be detected due to its dependency on the base point since $x_3 = x_P$ and $z_3 = z_P$ holds for each iteration of the Montgomery ladder.
6.7 Conclusions

Figure 6.14: Non-specific $t$-test results with fixed scalar: Profile 3 (10 000 traces)

![Graph](a) 1st-order $t$-test without random addresses  ![Graph](b) 1st-order $t$-test with random addresses

Figure 6.15: Non-specific $t$-test results with fixed scalar: Profile 4 (1 000 000 traces)

![Graph](a) 1st-order $t$-test without random addresses  ![Graph](b) 1st-order $t$-test with random addresses

Hence, we did not expect to prevent base-point-dependent leakage when only blinding the secret scalar.

**Profile 4: Combination**

The last profile again combines both countermeasures in order to analyze and evaluate the interaction of both mechanisms. In Figure 6.15 we present the evaluation results, again using 1 000 000 power traces. As we expect, we cannot observe first-order side-channel leakage for our Curve25519 architecture while both countermeasures are active.

### 6.7 Conclusions

In this chapter, we proposed two architectures for ECC supporting Diffie-Hellman key agreement and other cryptographic primitives based on Curve25519. Both architectures provide a security level comparable to AES-128 and process data at constant time to thwart timing attacks. We further included countermeasures with low resource overhead and a minor loss in performance to protect all cryptographic operations against SCA using the power consumption of a physical device (like SPA and DPA). Eventually, we demonstrated that our designs are capable of outperforming many previous works, both in hardware and software, despite of their moderate resource requirements and additional countermeasures against physical implementation attacks. Further, combining the full set of implemented SCA countermeasures protects our design against practical power analysis and prevents detectable, scalar-dependent nor base-point-dependent leakage even after capturing and analyzing 1000000 power measurements.
Chapter 7
Exploring RFC 7748: Curve448 with Side-Channel Protection

In this chapter, we demonstrate that the second candidate of RFC 7748, the elliptic curve Curve448 (also known as Ed448-Goldilocks), can indeed be efficiently and securely implemented in hardware. We present a novel architecture for Curve448 on a Xilinx XC7Z020 FPGA that can perform up to 1087 point multiplications per second at moderate cost of 1580 logic slices and 33 DSP units. Providing a second architecture including state-of-the-art side-channel protection mechanisms, the results presented in this chapter (based on a joint work with Tim Güneysu [SG17]) round off the portfolio of RFC 7748 and its realization in hardware.

Contents of this Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Introduction</td>
<td>73</td>
</tr>
<tr>
<td>7.2 Specification of Curve448</td>
<td>74</td>
</tr>
<tr>
<td>7.3 Design Rationales</td>
<td>76</td>
</tr>
<tr>
<td>7.4 Implementation Details</td>
<td>79</td>
</tr>
<tr>
<td>7.5 Results</td>
<td>81</td>
</tr>
<tr>
<td>7.6 Evaluation</td>
<td>83</td>
</tr>
<tr>
<td>7.7 Conclusion</td>
<td>87</td>
</tr>
</tbody>
</table>

7.1 Introduction

For current and future high-security applications operating at security levels beyond 128 bits (as provided by Curve25519), RFC 7748 introduces a second elliptic curve called Curve448 designed for a security level of 224 bits. Although both elliptic curves are defined over a prime field, the underlying finite field and its corresponding base field arithmetic is an essential difference of Curve25519 and Curve448 which prevents a simple adaption and reutilization of our architecture presented in Chapter 6. Instead of a Pseudo Mersenne prime as for Curve25519, Curve448 uses a Solinas prime which results in the need for a very different, more complex reduction scheme. In combination with the significantly larger field size, this requires a completely different design of the modular multiplication module to still fully benefit from (additionally
required) DSP units. In this context, we decided to augment the wider multiplier design of Curve448 to completely absorb the modular addition and subtraction operation in its data path, resulting in a single component for all modular operations. As a direct consequence, this single component design enables a simple memory architecture on the higher arithmetic level compared to the more complex butterfly architecture of Curve25519 to connect two separate components, each for modular addition, subtraction and multiplication. In conclusion, the larger field size and different base arithmetic created the need to a completely re-design of the entire ECC core.

7.1.1 Related work

Again, since ECC has become de-fact standard for public-key cryptography there exists a plethora of publications on hardware implementations that we cannot cover and discuss within the scope of this chapter. All the same, we want to briefly list the most relevant ones that were not mentioned in the previous chapter. However, for a detailed overview we would like to refer the interested reader to [dDQ07, FGM+10, FV12] once again.

As one of the first ECC architectures implemented on a FPGA, in 2001, Orlando and Paar [OP01] presented a design using pre-computations and Montgomery multiplications for improved performance results. Their work was followed by many more, including designs based on dedicated multipliers [ÔBPV03] and integrated DSP units embedded into modern FPGAs [GP08]. In the previous chapter, we presented an FPGA architecture for Curve25519, the first candidate of RFC 7748, based on results published in [SG14, SG15]. Besides, Järvinen et al. [JMAL16] recently presented a high-performance architecture for FOuRQ, a novel elliptic curve with about 128 bit security that supports highly-efficient point multiplications.

7.1.2 Contribution

In this chapter, we address this challenge and present the first hardware implementation of Curve448 tailored for modern Xilinx FPGA devices. Our work fills the gap of the previous chapter on Curve25519, based on work published in [SG14, SG15]. Despite of the similarity of Curve448 and Curve25519, we were faced with completely revising the design rationales due to specially-crafted parameters and the significantly increased field size of 448 bits. Interestingly, we can demonstrate that even in light of the asymptotically growing complexity in the field size, Curve448 still can provide similarly high throughput on a mid-range Xilinx XC7Z020 FPGA, providing a performance of more than 1000 point multiplications per second at moderate resource costs.

7.2 Specification of Curve448

Ed448-Goldilocks, since release of RFC 7748 better known as Curve448, was proposed and designed by Mike Hamburg [Ham15] in order to provide a high-security alternative (i.e., a field size between 384 and 521 bits) to existing NIST [LG09] and Brainpool curves [LM10]. The design process of Curve448 strictly heeded the SafeCurves policies and criteria [BL16] which should enable simple but secure implementations of an elliptic curve that meets all requirements.
7.2 Specification of Curve448

7.2.1 Field Arithmetic

Curve448\(^1\) is an untwisted Edwards curve, with a security level of 224 bits and defined by:

\[
\mathcal{E}_d : y^2 + x^2 \equiv 1 + dx^2y^2 \pmod{p},
\]

with factor \(d = -39\,081\), the Solinas prime \(p = 2^{448} - 2^{224} - 1\) and its golden ration \(\phi = 2^{224}\). Technically, Curve448 processes 448-bit values over \(\mathbb{GF}(p)\) but still is highly versatile and flexible for software implementations and different platforms (ranging from 8- to 64-bit) due to the fact that 448 = 56 × 8 = 28 × 16 = 14 × 32 = 7 × 64. Further, due to its golden ration \(\phi\), the Solinas prime allows fast and efficient Karatsuba-based multiplication of two operands \(A = (a_0 + a_1\phi)\) and \(B = (b_0 + b_1\phi)\), such that:

\[
C = A \cdot B = (a_0 + a_1\phi) \cdot (b_0 + b_1\phi) = (a_0b_0 + a_1b_1) + ((a_0 + a_1) \cdot (b_0 + b_1) - a_0b_0)\phi
\]

Eventually, modular inversion can be tweaked compared to inversions based on FLT using that, if \(p \equiv 3 \pmod{4}\), the Inverse Square Root (ISR) can be computed as \(\frac{1}{\pm \sqrt{x}} = x^{(p-3)/4}\), which directly leads to:

\[
x^{-1} = x \cdot \left(\frac{1}{\pm \sqrt{x^2}}\right)^2 = x \cdot \left[(x^2)^{(p-3)/4}\right]^2 = x^{p-2}
\]

7.2.2 Group Arithmetic

Similar to X25519, RFC 7748 presents the function X448 that performs a variable scalar-point multiplication on the Montgomery curve (Curve448) using the Montgomery ladder algorithm [Mon87] that combines point addition and point doubling in a single step. Again, given two points \(Q_1, Q_2 \in \mathcal{E}\), \(d = -39\,081\) and the difference \(Q_3 = Q_1 - Q_2\), a single iteration of the Montgomery ladder algorithm results in two points \(Q_4, Q_5 \in \mathcal{E}\) such that \(Q_4 = 2 \cdot Q_1 = (x_4, z_4)\) is the point doubling (see Equations 6.2 and 6.3) and \(Q_5 = Q_1 + Q_2 = (x_5, z_5)\) is the point addition (see Equations 6.4 and 6.5). Still, both computations involve 6 multiplications, 4 squarings, 4 additions and 4 subtractions over \(\mathbb{GF}(p)\) and are completely independent of the \(y\)-coordinate.

7.2.3 Point Multiplication

Given a public point \(P \in \mathcal{E}\) and a secret 448-bit scalar \(k\), the point multiplication routine computes another point \(Q = k \cdot P\) using a sequence of consecutive Montgomery ladder steps. For this purpose, the scalar is scanned bit-wise (starting from the most significant bit) and depending on the value of the currently processed bit, inputs to a single step of the Montgomery ladder algorithm are swapped. Hence, starting from the base point \(P\), the point at infinity \(O\) and their difference (i.e., \(P\)), the point multiplication finally yields the resulting point \(Q\) after the execution of 448 steps.

---

1According to RFC 7748, Curve448 represents a Montgomery curve and the untwisted Edwards curve actually is called Edwards448. However, since both curves are birationally equivalent, we use the term Curve448 synonymously throughout this work.
7.2.4 Side-Channel Protection

In order to prevent SCA attacks, modern cryptographic implementations usually encompass protection and countermeasures against various physical attacks. Fortunately, the application of the Montgomery ladder algorithm already provides basic and inherent protection against timing and SPA attacks, however, dedicated countermeasures against DPA attacks have to be added. Since there is a plethora of different countermeasures against DPA attacks, we refer the interested reader to an overview of Fan et al. [FGM+10, FV12], but only briefly summarize point randomization and scalar blinding as an example².

Point Randomization

Point randomization takes advantage of the additional degree of freedom introduced by the application of projective coordinate representations in order to randomize the representation of a base point using a randomly chosen \( \lambda \). During the base point transformation process (from affine to projective coordinates), the random factor is applied as shown in Equation 7.4, which yields in different point representations (depending on \( \lambda \)).

\[
P_r = (x_{P_r}, z_{P_r}) = (\lambda x_P, \lambda z_P) = (\lambda x_P, \lambda)
\]  

(7.4)

Fortunately, point randomization does not affect the correctness of the underlying scheme, as proven in Equation 7.5.

\[
x_{P_r} \cdot z_{P_r}^{-1} = \lambda x_P \cdot \lambda^{-1} = x_P \pmod{p}
\]  

(7.5)

Scalar Blinding

Instead of randomizing the base point \( P \), scalar blinding deals with the randomization of the second input parameter of the point multiplication operation that computes \( Q = k \cdot P \). Again, choosing a random factor \( r \), the original scalar \( k \) is blinded with a multiple of the group order \(|E|\), as shown in Equation 7.6.

\[
k_r = k + r \cdot |E|
\]  

(7.6)

Given the blinded scalar \( k_r \), correctness of the underlying scheme is proven by the fact that the product of group order and base point returns the point at infinity. Hence, the point multiplication still results in \( Q \) as shown in Equation 7.7.

\[
k_r \cdot P = (k + r \cdot |E|) \cdot P = k \cdot P + r \cdot O = k \cdot P
\]  

(7.7)

7.3 Design Rationales

Hardware implementations of modern cryptographic primitives can be designed in many ways and optimized from different perspectives, such as performance and throughput, resource utilization and hardware footprint, physical and side-channel security and many more. In this section we discuss several optimization strategies and different solution approaches in order to achieve the chosen objectives for our final optimized implementation.

²We have chosen this set of countermeasures since it provides us with the opportunity to randomize and protect all input parameters of a single point multiplication.
7.3 Design Rationales

7.3.1 Implementation Objectives

To the best of our knowledge, this is one of the first hardware implementations of an elliptic curve with security level of more than 128 bits, so we can mainly compare to implementations targeting lower security requirements which are naturally faster. However, implementing high-security ECC must not prevent us from providing a competitive hardware architecture, so we decided to primarily optimize our design in terms of maximum performance and throughput. Still, other implementation objectives such as resource utilization and physical security should not be ignored but only play a secondary role.

7.3.2 General Optimization Strategies

In order to increase the performance of our design to a maximum we applied several optimization strategies and evaluated different design choices. In general, critical path minimization and parallelization are among the most common approaches that allow to increase the throughput of hardware architectures. Hence, given a single instance of our design, we put a lot of engineering effort to minimize the critical path and increase the maximum frequency by hand while maintaining a balanced resource utilization in order to allow the implementation of multiple instances in parallel (multi-core design). In the following, we provide some details on our optimization strategies and approaches.

**Register Balancing.** It is a common approach to use additional register stages to minimize the critical path delay of hardware architectures. In particular for FPGA implementations, this is the straightforward solution since registers are placed after every logic element and would be wasted if they remain unused. Still, logic elements have to be balanced across the entire design in order to avoid delays due to critical data dependencies.

**Parallelization.** Hardware implementation particularly can benefit from parallelization techniques which can be applied on various levels of the design hierarchy. In particular, a modular implementation enables independent operation of sub-components which are separated by register stages. Eventually, placing multiple instances of a cryptographic primitive on the same device can give further performance boosts only limited by the available resources of the physical device.

**Resource Sharing.** Naturally, faster designs and better performance can be purchased with more resources, but sharing may allow reusing existing components instead of implementing additional modules. Similar to parallelization, resource sharing can be applied at different levels of the design hierarchy, i.e., within a single but also among several instances (multi-core design).

**Data Path Expansion.** In order to avoid idling phases of sub-modules working in parallel, the internal data path of the design has to be expanded as well in order to provide each component with data on time. Since this naturally will increase the area utilization of the architecture, designers have to find the best trade-off between either a serial and low-area or a parallel and low-latency design.
Chapter 7 Exploring RFC 7748: Curve448 with Side-Channel Protection

7.3.3 Field Arithmetic Optimizations

According to Section 7.2.1, all operations on elliptic curves over $GF(p)$ are based on operations over finite fields, such as modular addition, subtraction, multiplication, and inversion. Hence, the fundamental component of our hardware architecture is the Field Arithmetic Unit (cf. Section 7.4.1) which has to be designed and optimized thoroughly before implementation since its operation is crucial and will impact to a great extent the final performance. Due to this fact, we had to do the entire optimization manually without relying on automated tools. In the following, we present and discuss several design choices leading to our high-performance Field Arithmetic Unit in particular optimized for Curve448.

Modular Inversion

In general, hardware designers have to decide whether performing modular inversion using dedicated units, e.g., based on the binary EEA, or re-using modular multiplication to compute the inversion using FLT or the ISR tweak. We decided against a dedicated unit due to its...
bad ratio between performance gain and area increase (for a single core), instead we intend to perform modular inversion using the ISR tweak in Equation 7.3.

Modular Multiplication

With regard to the modular multiplication, we investigated and compared two different approaches: Karatsuba Multiplication and Schoolbook Multiplication with Interleaved Reduction.

Karatsuba. The Karatsuba algorithm, as shown in Equation 7.2, uses decomposition in order to perform multiplications in partial steps which are more favorable due to smaller operand sizes. This approach allows to reduce the size of the multiplication unit (in terms of occupied area) but at cost of additional latency and control overhead.

Schoolbook. Due to the Solinas prime, the common schoolbook multiplication can be combined with an interleaved reduction step based on the fact that $2^{448} \equiv 2^{224} + 1 \pmod{p}$. As shown in Figure 7.1, each multiplication step is interleaved by a reduction before the partial products are accumulated.

Since the primary goal of this work is a high-performance point multiplication, we decided to build a modular multiplication unit based on the Schoolbook Multiplication with Interleaved Reduction. This approach has a lower latency compared to the Karatsuba Multiplication and more important, it perfectly maps to the available hardware resources (i.e., the DSP units) as shown in Section 7.4.1. Note, however, that both approaches need a final processing and reduction step which is handled by another, dedicated reduction unit.

Architectural Customizations

In Section 7.3.2 we provide a general set of optimization strategies that we considered throughout the entire design process. Based upon this, we decided to extend the internal data path of our architecture to a full width of 448 bits in order to allow maximum parallelization and optimal performance within all sub-modules. In general, this allows to provide the Field Arithmetic Unit (FAU) with all operands on time avoiding or reducing idling phases due to load or store operations.

Further, since most of the time, our FAU will perform modular multiplications and each multiplication is performed in 28 steps, thus requiring at least 28 cycles, we implement the optimal number of additional pipeline stages within the DSP units (i.e., four) in order to allow operation at maximum frequency. Note, however, that this modification as well increases latency of modular additions and subtractions, since we share and re-use the DSP units for these operations.

Eventually, our FAU is divided into sub-modules which are separated by register stages in order to allow independent operation and in order to decrease the critical path.

7.4 Implementation Details

In this section, we provide implementation details of our CURVE448 hardware architecture, in particular the arithmetic units of our design. In addition, we provide details on an enhanced version of our basic architecture which provides further protection mechanism and countermeasures
against side-channel attacks. Most of all, we like to point out that this additional protection can be integrated into the basic architecture with only minimum effort and overhead.

7.4.1 Field Arithmetic Unit

As mentioned earlier, the FAU (as sketched in Figure 7.2) is the basic component of our architecture. In principle, it can be subdivided into two different components, an Arithmetic Core (AC) and a Reduction Core (RC), that we explain in detail in the following.

Arithmetic Core (AC). In order to perform arithmetic operations, the body of our AC is implemented using 28 fully pipelined DSPs in parallel. Each DSP provides different configurations, such as 16-bit addition, 16-bit subtraction or $16 \times 16$-bit multiplication. By selecting the according configuration during run time, the AC implements full data path additions, subtractions and multiplications. Hence, sharing the DSPs allows implementing Carry-Save Additions and Subtractions as well as Schoolbook Multiplications with Interleaved Reduction within a single unit. The final accumulation and reduction of the results is performed by the RC.

Reduction Core (RC). The RC consists of another five DSPs, three to preadd partial products and two to accumulate the carries and to perform the final reduction. In general, the reduction process is performed serially on 17-bit words in order to take advantage of the
internal shift operation of the DSPs. However, due to the serial design, the reduction is the bottleneck of the Field Arithmetic Unit. Still, both, the AC and the RC, can be operated independently and in parallel in order to alleviate the disadvantage of the serial design.

### 7.4.2 Side-Channel Countermeasures

In this section we explain the necessary modifications of our architecture in order to provide a basic protection against side-channel attacks using point randomization and scalar blinding.

**Point Randomization**

As explained in Section 7.2.4, point randomization uses a randomly chosen parameter $\lambda$ which is applied to the base point. In general, this process initializes the $z$-coordinate with $\lambda$ and uses a modular multiplication in order to update the $x$-coordinate. Hence, this countermeasure can be integrated easily, mainly by using an additional multiplication call during the initialization phase of the implementation.

**Scalar Blinding**

According to Section 7.2.4, the scalar is blinded by a random multiple of the group order and due to the special structure of the underlying prime, it is advisable to use random factors of at least half of the field size [SW15]. Hence, using two additional DSP units, the secret scalar is blinded and expanded by 221 bits. In particular, we opted to use a blinding factor of $221 = 13 \times 17$ bits, since it perfectly fits to the multiplier within the DSP that can perform $24 \times 17$-bit signed multiplications. However, based on the increased scalar size, the final performance of the implementation naturally drops due to additional Montgomery steps that have to be performed.

### 7.5 Results

In this section, we present implementation and performance results after place-and-route (PAR) for both designs and provide a comparison to related work. Note, that all results were obtained for a Xilinx XC7Z020CLG484-3 FPGA using the Vivado Design Suite 2015.4.

#### 7.5.1 Implementation Results

Table 7.1 provides the hardware area results of both designs (with and without side-channel countermeasures) in terms of occupied resources. Obviously, the limiting factor for both designs (considering maximum parallelization and multi-core architectures) is the number of DSP that are available. However, although the resource utilization slightly increases for the protected design, both variants allow to place up to six cores in parallel (on the chosen mid-range Xilinx Zynq FPGA).

Table 7.1 summarizes the performance results for both designs broken down to the group and field arithmetic operations. Since the unprotected design can achieve a slightly higher maximum clock frequency (357 MHz) compared to the protected design (335 MHz), operations can be
performed faster although the number of clock cycles does not change for the field arithmetic and group operations. However, due to the increased bit-size of the blinded scalar, the final point multiplication of the protected design involves additional steps of the Montgomery ladder algorithm, hence the overall latency (in terms of clock cycles) increases by about 45%.

### 7.5.2 Comparison

To the best of our knowledge, there are only few implementations of ECC schemes with a security level of above 128 bits available to the public and academic community that are tailored for FPGA devices. Along with the fact that modern FPGA architectures have evolved and changed dramatically, this renders a fair, meaningful discussion and comparison of different designs and implementations hardly possible. However, we still would like to put our results in the context of existing implementations and compare to different architectures – but of course with a note of warning.

Table 7.2 list two FPGA architectures of ECC schemes over prime fields, i.e., Curve25519 (second candidate of RFC 7748) and an implementation of an elliptic curve called FOURQ, recently presented at CHES 2016. Note, however, that all these implementations target a lower level of security (123-127 bits) which makes fair comparisons rather difficult. Still, we would like to highlight some cutting features of our design in comparison to the related work.

Given the fact, that the time complexity is assumed to grow cubic in the field size, we expect a final performance of 15-20% in terms of operations per second in comparison to the implementations listed in Table 7.2. All the same, our implementation mostly exceeds the expectations which emphasizes its highly optimized architecture, although, naturally, more resources are occupied.
### 7.6 Evaluation

Table 7.2: Comparison of different designs for Elliptic Curve Cryptography over prime fields on FPGAs

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Device</th>
<th>Security</th>
<th>Architecture</th>
<th>Performance</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Data Path</td>
<td>LS</td>
<td>DSP</td>
<td>BRAM</td>
</tr>
<tr>
<td>NIST P-224</td>
<td>XCC4VFX12</td>
<td>112 bit</td>
<td>32 bit</td>
<td>1.580</td>
<td>26</td>
</tr>
<tr>
<td>single-core</td>
<td>XCC4VFX12</td>
<td>112 bit</td>
<td>32 bit</td>
<td>24.52</td>
<td>468</td>
</tr>
<tr>
<td>multi-core</td>
<td>XCC4VFX12</td>
<td>128 bit</td>
<td>32 bit</td>
<td>21715</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>XCC4VFX12</td>
<td>128 bit</td>
<td>32 bit</td>
<td>24574</td>
<td>512</td>
</tr>
<tr>
<td>NIST P-256</td>
<td>XCC4VFX12</td>
<td>128 bit</td>
<td>32 bit</td>
<td>21715</td>
<td>32</td>
</tr>
<tr>
<td>single-core</td>
<td>XCC4VFX12</td>
<td>128 bit</td>
<td>32 bit</td>
<td>24574</td>
<td>512</td>
</tr>
<tr>
<td>multi-core</td>
<td>XCC4VFX12</td>
<td>128 bit</td>
<td>32 bit</td>
<td>24574</td>
<td>512</td>
</tr>
<tr>
<td>FOURQ</td>
<td>XC7Z020</td>
<td>123 bit</td>
<td>34 bit</td>
<td>1029</td>
<td>20</td>
</tr>
<tr>
<td>single-core3</td>
<td>XC7Z020</td>
<td>123 bit</td>
<td>34 bit</td>
<td>1169</td>
<td>22</td>
</tr>
<tr>
<td>single-core4</td>
<td>XC7Z020</td>
<td>123 bit</td>
<td>34 bit</td>
<td>1169</td>
<td>22</td>
</tr>
<tr>
<td>multi-core4</td>
<td>XC7Z020</td>
<td>123 bit</td>
<td>34 bit</td>
<td>1169</td>
<td>22</td>
</tr>
<tr>
<td>Curve25519</td>
<td>XC7Z020</td>
<td>127 bit</td>
<td>34 bit</td>
<td>11277</td>
<td>220</td>
</tr>
<tr>
<td>single-core1</td>
<td>XC7Z020</td>
<td>127 bit</td>
<td>34 bit</td>
<td>10903</td>
<td>220</td>
</tr>
<tr>
<td>single-core2</td>
<td>XC7Z020</td>
<td>127 bit</td>
<td>34 bit</td>
<td>10903</td>
<td>220</td>
</tr>
<tr>
<td>multi-core1</td>
<td>XC7Z020</td>
<td>127 bit</td>
<td>34 bit</td>
<td>10903</td>
<td>220</td>
</tr>
<tr>
<td>multi-core2</td>
<td>XC7Z020</td>
<td>127 bit</td>
<td>34 bit</td>
<td>10903</td>
<td>220</td>
</tr>
<tr>
<td>Curve448</td>
<td>XC7Z020</td>
<td>224 bit</td>
<td>44 bit</td>
<td>1580</td>
<td>33</td>
</tr>
<tr>
<td>single-core1</td>
<td>XC7Z020</td>
<td>224 bit</td>
<td>44 bit</td>
<td>1648</td>
<td>35</td>
</tr>
<tr>
<td>single-core2</td>
<td>XC7Z020</td>
<td>224 bit</td>
<td>44 bit</td>
<td>1648</td>
<td>35</td>
</tr>
</tbody>
</table>

| 1 unprotected, 2 protected, 3 Montgomery Ladder, 4 Endomorphism |

#### 7.6 Evaluation

Eventually, we performed a practical, test-based Side-Channel Analysis and leakage assessment in order to evaluate our implementation and included countermeasures. To this end, we performed practical power measurements on a SAKURA-X [Lab] side-channel evaluation board. Our design was running on a 7-series Kintex FPGA (XC7K160T) using a stable, jitter-free 96 MHz clock signal. More precisely, we measured the voltage drop over an $1 \Omega$ resistor in the $V_{dd}$ path by means of a LeCroy digital oscilloscope (HDO6054) using a sampling rate of 250 MS/s and 20 MHz bandwidth limitation.

#### 7.6.1 Detection of Scalar-Dependent Leakage

During our evaluation and leakage assessment process we first focus on the detection of scalar-dependent leakage, i.e., the core is provided with a fixed base point ($x_P = 5$) throughout the entire evaluation process while the scalar is chosen from a set of random scalars. In order to apply a non-specific $t$-test, the measurements are randomly interleaved with operations based on a fix scalar. Further, in order to avoid any input dependent leakage we slightly modified our architecture such that the scalar blinding and coordinate system transformation have to be performed externally while the core only accepts masked inputs, i.e., blinded scalars and points in projective coordinate representation. However, the final result of the scalar-point multiplication still will be available only in an unprotected state which certainly results in...
output dependent leakage. Hence, to avoid such result-depending detection of leakage, we exclude the final operations after the inversion from our observations.

In general, we define four different profiles in order assess our setup and countermeasures individually as well as in combination. In particular, Profile 1 provides a reference since all countermeasures are disabled, i.e., the PRNG does not provide random values. Profile 2 and 3 then investigate the scalar blinding and coordinate randomization countermeasure individually before Profile 4 evaluates our fully protected design with both countermeasure active in parallel.

**Profile 1: PRNG off**

The first profile provides a reference measurement in order to investigate the correctness of our setup and evaluation process. In that sense, we disabled our PRNG that provides the randomness for our countermeasures, i.e., the scalar is not blinded but only padded with zeros and the base point is provided in plain (i.e., $z_p = 1$). In Figure 6.7(a), we provide a mean trace derived from 1000 measurements where we clearly can distinguish the initial processing of the zero padding before the unprotected scalar is processed and the inversion is performed. Further, in Figure 7.4(a), we provide first-order $t$-test results after capturing 10,000 power traces. In addition, Figure 7.4(b) presents the evolution of the absolute maximum of the $t$-statistic clearly indicating that with increasing measurements the confidence for detected leakage increases accordingly.

**Profile 2: Randomized Projective Coordinates**

During our second measurement profile, we provide our core with the base point in randomized projective coordinate representation. However, according to Figure 7.5(a), we still can observe scalar-dependent leakage particularly after the zero-padding of the scalar has been processed. Although the $t$-value is smaller that for our first profile (even after processing 100,000 power traces), we still can observe leakage with high confidence that steadily increases with growing number of processed traces (see Figure 7.5(b)).

**Profile 3: Scalar Blinding**

Our third profile investigates the effect and impact of the scalar blinding countermeasure. Figure 7.6 provides the evaluation results after 100,000 measurements. Although the $t$-value occasionally exceeds the threshold of 4.5, it does not increase significantly over the number of processed traces (cf. Figure 7.6(b)). In particular, only few points (that even vary over time) exceed
the threshold indicating that we detected only false positives. To this end, we can conclude that the scalar blinding countermeasure does not exhibit any detectable leakage and we cannot distinguish the processing of randomly chosen and fixed scalars for the given number of measurements.

Profile 4: Combination

Eventually, our last evaluation profile combines both countermeasures (scalar blinding and coordinate randomization) in order to assess the security of our fully protected core. First of all, Figure 7.3(b) provides a mean trace over 1000 measurements with both countermeasures enabled. Obviously, we cannot distinguish different parts of the scalar (due to its blinded state) as for our first profile. Further, we cannot detect any scalar-dependent leakage even after processing 1 000 000 power traces as can be seen in Figure 7.7(a) although we again detect some false positives over the number of measurements that exceed the threshold of 4.5 (see Figure 7.7(b)).

7.6.2 Detection of Base-Point-Dependent Leakage

In a second approach, we focused our analysis on the detection of base-point-dependent leakage where we used a fixed scalar while the base point was chosen from a random set randomly interleaved with the default base point \( x_P = 5 \). Again, to ensure correctness in particular for the group order of the (randomly) chosen base point, we modified our measurement setup similar to the evaluation of CURVE25519. To this end, the random base point \( P' \) is chosen depending on \( P \) as \( P' = s \times P \), where \( s \) is a randomly chosen scalar and the additional scalar-point multiplication is performed in software prior to the communication with our implemented core. In addition, the secret scalar has been fixed to the test value that is given in RFC 7748.
Profile 1: PRNG off

As usual, the first setup provides a reference in order to check the measurement setup. Figure 7.8 provides the evaluation results for this profile using 10,000 measurements. As expected, we can detect side-channel leakage with high confidence that is still increasing with the number of traces.

Profile 2: Randomized Projective Coordinates

Figure 7.9 provides the evaluation results for our second profile, using 100,000 measurements while the coordinate representations for the base point has been randomized. Surprisingly, we detect first-order side-channel leakage during the operation of our Curve448 core. However, the detected leakage only appears during the first 224 iterations of the Montgomery ladder algorithm, i.e., those iterations where the secret scalar has been padded with leading zeros. Hence, we expect to prevent this leakage by using a randomized representation of the scalar (scalar blinding) to avoid leading zeros or by dispense with the padding.

Profile 3: Scalar Blinding

Our third profile evaluates the individual effects of the scalar blinding countermeasure on base-point-dependent leakage. In Figure 7.10 we provide the evaluation results after using only 10,000 measured power traces. Obviously, in contrast to the third profile while using a fixed base point, we can clearly observe first-order side-channel leakage with high confidence even increasing in the number of traces. However, since the internal processing of $Q_3 = Q_1 - Q_2$ is directly related to the base point, i.e., $x_3 = x_P$ and $z_3 = z_P$, we can detect any operation that involves $Q_3$. More precisely, since $Q_3$ is constant throughout each point multiplication and used
during every iteration of the Montgomery ladder algorithm, we did not expect to prevent the detection of side-channel leakage solely relying on the blinding of the secret scalar.

Profile 4: Combination

Eventually, the last profile again evaluates the interaction and combination of both countermeasures in order to protect our design against first-order SCA. Figure 7.11 shows our evaluation results using 1,000,000 measured power traces. As expected, the combination of the randomized coordinate representation along with the scalar randomization avoids the zero-padding of the secret scalar, hence prevents the leakage as detected in Profile 2. To this end, we cannot observe any first-order leakage while both countermeasures are active and combined even when using one million measurements which confirms the correctness and effectiveness of our implemented countermeasures.

7.7 Conclusion

In this chapter, we present two new architectures for ECC based on Curve448, the second elliptic curve candidate of RFC 7748. Although Curve448 mainly was designed for efficient implementations on various software platforms, we demonstrate that it perfectly maps to existing FPGA architectures as well. Further, we show that our implementation can be easily enhanced with protection mechanisms against side-channel attacks. Whereas our unprotected allows up to 1087 point multiplications per second on a mid-range Xilinx XC7Z020 FPGA, our protected design still achieves a performance of 708 operations per second, what is expected to be sufficient even for most high-performance applications. In addition, our fully protected
design does not exhibit any detectable, scalar-dependent nor base-point-dependent leakage even after processing 1 000 000 power measurements.
Part III

Hardware Agility using Dynamic Reconfiguration
Chapter 8
Dynamic Logic Reconfiguration

Reconfigurability is a unique feature of modern FPGAs devices to load and update hardware circuits just on demand. This also implies that a completely different set of circuits might operate at the exact same location of the FPGA at different time slots, making it difficult for an external observer or attacker to predict what will happen at what time. In this part, we discuss the application of inherent local reconfiguration potential of modern FPGAs, in particular the Distributed Memory features of recent Xilinx devices, for the construction of novel and dynamic side-channel countermeasures. To this end, the following chapter, which is based on a joint work with Amir Moradi, Oliver Mischke, and Tim Güneysu [SMMG15c], presents and evaluates a novel hardware implementation of the lightweight cipher PRESENT with built-in side-channel countermeasures based on local and dynamic logic reconfiguration. In our proposed design we make use of CFGLUTs as Distributed Memory to nearly instantaneously change hardware internals of our cipher implementation in order to improve resistance against side-channel attacks. More precisely, we provide evidence from practical experiments based on a Spartan-6 platform that with even 10 million recorded power traces we are unable to detect first-order leakage using a specific t-test as leakage assessment methodology.

Contents of this Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Introduction</td>
<td>91</td>
</tr>
<tr>
<td>8.2 Preliminaries</td>
<td>92</td>
</tr>
<tr>
<td>8.3 Countermeasures</td>
<td>94</td>
</tr>
<tr>
<td>8.4 Practical Evaluation</td>
<td>96</td>
</tr>
<tr>
<td>8.5 Conclusion</td>
<td>100</td>
</tr>
</tbody>
</table>

8.1 Introduction

The theoretic security, provided by cryptographic primitives implemented on physical devices, can be easily overcome e.g., by inspecting their power consumption and exploiting information leakage of related device internals through SCA. Common countermeasures, typically classified as masking and hiding [MOP07], seek to randomize intermediate values and reduce the overall SNR e.g., by power equalization [TV04] or noise generation [MOP07, GM11].
Although various countermeasures have been proposed and analyzed over time, implementations still require a strong intuition and engineering skills to achieve the claimed level of security due to manifold pitfalls. In particular, the presence of glitches inside combinatorial circuits (for example see [MPO05, MME10]) can impair the security of masking countermeasures. Hence, instead of using combinatorial circuits, critical elements of cryptographic primitives such as S-boxes can be realized as look-up tables that are dynamically randomized and updated in memory. In [GM11], such an approach based on scrambling BRAM content of FPGAs for an T-box-based AES implementation has been presented.

8.1.1 Contribution

In this chapter, we present a novel implementation approach to randomize look-up tables which makes use of the local, dynamic reconfiguration feature of certain building blocks of FPGAs. In particular, our approach employs special LUT elements inside modern Xilinx FPGAs as CFGLUT [Xil13] that can change their configuration on-the-fly. This not only allows a more fine-grain but also by far a more flexible randomization of look-up tables on FPGAs compared to the previous approach of BMS using large and static 18 kBit BRAMs.

As a case study we applied the new technique for dynamically randomized look-up tables on an implementation of the PRESENT cipher [BKL+07]. In addition, we decomposed each S-box into several stages and utilized register precharge to increase the overall resistance against SCA attacks. We tested the effectiveness of each countermeasure and their different combinations by applying the TVLA methodology proposed by Goodwill et al. [GJJR11]. With all countermeasures enabled, we are unable to detect a first-order leakage on 10 million captured power traces from an SCA evaluation platform populated with a Spartan-6 FPGA. This significantly improves the results from [MGV08] which is the only previous work that deals with using dynamic (partial) reconfiguration as SCA countermeasure.

8.2 Preliminaries

In the following, we briefly describe the structure and function of CFGLUTs and how any generic n-input and m-output Boolean logic function can be realized based on our Reconfigurable Function Table (RFT) concept before, we introduce our basic architecture of the lightweight block cipher PRESENT.

8.2.1 Configurable Look-Up Table

Starting with the Virtex-5 device family, Xilinx devices are equipped with elements named Configurable Look-Up Table that enable fast dynamic logic reconfiguration during run time. Instead of modifying the configuration by full or partial bitstream reconfiguration, the function of these configurable LUTs can be loaded and replaced instantaneously without external intervention. Internally, each CFGLUT consists of a distributed LUTRAM block that can operate as a 16-bit Shift Register Look-Up Table (SRL16). Hence, even for older devices that do not provide dynamic logic reconfiguration, SRL16 instances can be used with slight effort to achieve the same functionality.
Figure 8.1 outlines the internal structure of a Configurable LUT that is composed of a 16-bit shiftable configuration memory and a subsequent multiplexer stage. Due to the restricted configuration memory size of 16 bits, each CFGLUT can realize at most a 4-input and 1-output Boolean function. In order to implement any \((n \times m)\) Boolean function, we now introduce the general concept for Reconfigurable Function Tables.

### 8.2.2 Reconfigurable Function Table

In general, any \((n \times 1)\) Boolean function can be realized using multiple configurable LUTs combined by a multiplexer cascade. If this structure is instantiated \(m\) times with shared inputs, we call this a \((n \times m)\) RFT. Internally, each RFT consists of \(m \cdot \lceil 2^n - 4 \rceil\) CFGLUT instances structured as shown in Figure 8.1. Using the example of an AES S-box, the structure of an \((8 \times 8)\) RFT can be illustrated as: an 8-input, 1-output Boolean function can be realized using 16 CFGLUTs connected by cascading 15 multiplexers (each 2 : 1)\(^1\). This structure is instantiated 8 times while sharing the input signals. Thus, the entire AES S-box RFT could be built of 128 CFGLUT. Obviously, for large RFTs this structure becomes inefficient since e.g., a non-reconfigurable AES S-box can be realized by 32 LUT-6 (excluding the necessary multiplexers) compared to 128 CFGLUT which can be mounted into 64 LUT-6. Therefore, we primarily propose RFTs for lightweight ciphers or those employing \((4 \times 4)\) S-boxes that can be realized more easily by a RFT of 4 CFGLUTs. To this end, we have chosen to implement the PRESENT cipher as a case study in this chapter.

### 8.2.3 Basic Architecture

PRESENT is a symmetric lightweight block cipher with a block size of 64 bits. The encryption scheme is based on a SPN encrypting a plaintext within 31 rounds using 32 sub-keys. Each sub-key is derived from an initial 80-bit (or 128-bit) key. Figure 8.2 provides an overview of our architecture design implemented on an FPGA. We opted to implement the PRESENT encryption scheme in a round-based architecture that requires two clock cycles per round and derives sub-keys on-the-fly. The data path has a width of 64 bits and the substitution layer

\(^1\)This overhead can be reduced by utilizing LUTs as multiplexer.
consists of 16 parallel S-boxes including the state registers. The permutation is applied bit-wise and can be realized in hardware by plain routing resources. The total overhead of our protected design compared to a simple round based architecture is shown in Table 8.1.

### 8.3 Countermeasures

The evaluation of hardware countermeasures is a tricky problem due to the many different reasons for side-channel leakages in the design space, including the routing and placement of resources. However, in order to fairly investigate and compare the effectiveness of our proposed countermeasures, we apply a modular approach. More precisely, each countermeasure can be enabled separately for individual and joint evaluation with others. We now delve into the details how we integrated the countermeasures into the basic architecture presented in Section 8.2.3.

#### 8.3.1 Random S-box Decomposition

Side-channel leakages can often be found in power traces on critical transitions in the combinatorial circuit after a change in a driving register. Since most side-channel attacks on symmetric block ciphers target the output of the nonlinear substitution layer, it might be beneficial to avoid the storage of the S-box outputs into such registers. Therefore, we introduce the idea of a random S-box decomposition. By moving the state register into the substitution layer, we split the standard S-box up into two (random) mappings. Hence, we never store a correct S-box output into a register but only (randomly) mapped values.

The two mappings, surrounding the state register, are built of two \((4 \times 4)\) RFTs in order to be able to update their configuration for every encryption. The first RFT is configured to realize a randomly selected bijection \(R_1\). Hence, the second RFT should be configured to implement the bijection \(R_2\) in such a way that \(\forall x, R_2(R_1(x)) = S(x)\). This means that only the application of both mappings results in the correct S-box output which, however, is never stored to a register. Instead, the intermediate register only holds a value \(R_1(x)\) which is unpredictable for an attacker.

The configuration of both RFTs (of \(R_1\) and \(R_2\)) is computed randomly prior to every encryption within 16 clock cycles. \(R_1\) is computed by swapping two random elements of the identity configuration while \(R_2\) is computed using the equation \(R_2(R_1(x)) = S(x)\). All 16 S-boxes of the round function share the same configuration and before the encryption of a plaintext can
Table 8.1: Comparison of resource utilization and time overhead for unprotected and protected PRESENT designs

<table>
<thead>
<tr>
<th>Design Component</th>
<th>Unprotected</th>
<th>Protected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Logic (LUT)</td>
<td>Memory (FF)</td>
</tr>
<tr>
<td>Keyschedule</td>
<td>48</td>
<td>85</td>
</tr>
<tr>
<td>Round Function</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>Key Addition</td>
<td>64</td>
<td>0</td>
</tr>
<tr>
<td>Single S-box</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Countermeasure</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Decomp. + Masking</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Precharge</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Performance**

<table>
<thead>
<tr>
<th>Latency</th>
<th></th>
<th>Latency</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S-box Decomposition</td>
<td>— 0 cycles —</td>
<td>— 16* cycles —</td>
<td></td>
</tr>
<tr>
<td>Reconfiguration</td>
<td>— 0 cycles —</td>
<td>— 16 cycles —</td>
<td></td>
</tr>
<tr>
<td>Encryption</td>
<td>— 31 cycles —</td>
<td>— 62 cycles —</td>
<td></td>
</tr>
</tbody>
</table>

*This time can be avoided if a new decomposition is computed in parallel to a previous encryption.*

start, the derived configuration is loaded into the RFTs within another 16 clock cycles. It is possible to compute new configurations in parallel to an encryption, but to evaluate our design in a worst-case scenario (i.e., the best an attacker can achieve) we avoided such parallelism.

Note that if S-box decomposition is disabled, the configuration of \( R_1 \) is the standard PRESENT S-box, and hence \( R_2 \) realizes the identity mapping.

### 8.3.2 Boolean Masking

Algorithmic Boolean masking is intended to randomize the intermediate values of a cipher implementation by additively introducing random masks. Since the modification of intermediate values affects the computation, Boolean masking requires some adaptations for the implementation. In particular, the nonlinear substitution layer, i.e., the S-box configuration of our cipher implementation, has to be updated depending on the randomly selected mask. In order to keep the masking countermeasure compatible to the S-box decomposition, we mask both RFTs (\( R_1 \) and \( R_2 \)) independently based on \( m_1 \) and \( m_2 \) respectively. To keep the masked tables constant for every round we update the masks only prior to each encryption. The configuration of each table, i.e., \( R_1 \) and \( R_2 \), is then recomputed as follows:

\[
R'_1(x) = R_1(x \oplus m_1) \oplus m_2 \\
R'_2(x) = R_2(x \oplus m_2) \oplus P^{-1}(m_1),
\]

where \( P \) denotes the PRESENT permutation layer. This means that each nibble of the round state is masked by a 4-bit mask \( m_1 \). After the computation of the first mapping \( R_1 \) (regardless of whether \( R_1 \) is a random mapping or the standard S-box), the mask is changed to \( m_2 \) and the masked state is stored to the register stage. Afterwards, the second mapping is applied, changing the mask for every nibble from \( m_2 \) to the inverse permutation of \( m_1 \). This means that after the linear permutation layer of PRESENT, the round state is again masked with the initial mask \( m_1 \). Thus, the masking is kept constant for all rounds of one encryption.
We need to emphasize that we do not reuse any masks at each round. In other words, for each state nibble, two independent 4-bit $m_1$ and $m_2$ are chosen from a uniform distribution. In total, for the masking our design requires 128 random bits prior to each encryption. The masking countermeasure can be disabled easily by setting all $m_1$ and $m_2$ to zero.

### 8.3.3 Register Precharge

Since in our design the masks do not change during one encryption, the round inputs (resp. outputs) are masked with the same masks. Hence, the state register introduces a leakage by storing the consecutive round inputs. Such a leakage can be easily detected by a Hamming distance model as

$$ \text{HD}(x \oplus m, y \oplus m) = \text{HW}(x \oplus y). $$

Therefore, we expanded the single register stage into two registers to avoid such a leakage. Depending on the initial content of the registers, the encryption rounds are always interleaved with another dummy encryption round. This technique avoids the aforementioned leakage but reduces the throughput by the factor of two. If this feature is disabled, the loop multiplexer (see Figure 8.2) passes the plaintext for two initial clock cycles, filling both registers with the same value. The final design of our S-box, including all proposed countermeasures is shown in Figure 8.3. For our implementation, this S-box is instantiated 16 times as depicted in Figure 8.2.

### 8.4 Practical Evaluation

We employed a SAKURA-G platform [GIS14], i.e., a Spartan-6 FPGA, for practical SCA evaluations. The power consumption traces have been measured by means of a *pico Technology* digital oscilloscope (PicoScope 6402B) by monitoring the voltage drop over a 1Ω resistor in the $V_{dd}$ path. We have used the embedded amplifier of the SAKURA-G and recorded the traces at a sampling rate of 625 MS/s while the design was running at a low clock frequency of 3 MHz to reduce noise caused by the overlap of power traces.
8.4 Practical Evaluation

8.4.1 No Countermeasure

For comparison purposes we start our evaluation with a reference measurement, i.e., Profile 0, measuring one million encryption runs with random plaintexts and a fixed key as input and each of our proposed countermeasures disabled. Figure 8.4 shows a sample power trace, where all 31 rounds, each taking 2 clock cycles, are clearly distinguishable. The results after the application of the specific statistical \( t \)-test for all 144 models are shown in Figures 8.5(a), 8.5(b), 8.5(c), and 8.5(d) respectively. Obviously, for these measurements, the \(|t|\) value exceeds the threshold of 4.5 for all models which means that all tests detect a first-order leakage.

8.4.2 Single Countermeasure

In the next step, we investigate all countermeasures insulated in order to assess the impact of each of them on the first-order leakage of our design. Therefore, the following three measurement profiles are evaluated:

Profile 1: Random S-box Decomposition. This countermeasure was introduced to avoid buffering intermediate S-box values. The results of all applied tests, i.e., all 144 models, can be seen in Figures 8.6(a), 8.6(b), 8.6(c), and 8.6(d) respectively. Although the S-box decomposition avoids the S-box output to be stored in a register, it is computed by the RFT of \( R_2 \), and its associated leakage is still detectable. According to the shown results, random S-box decomposition reduces the first-order leakage but is not sufficient to be solely applied.

Profile 2: Boolean Masking. Masking of intermediate values aims at decorrelating power consumption and intermediate values using randomization of the processed values. The evaluation of this countermeasure based on 1 million power traces is shown in Figures 8.7(a), 8.7(b), 8.7(c), and 8.7(d). Due to the reuse of masks for all rounds, the Hamming distance between two consecutive values does not depend on the masks but only on the round outputs. Therefore, the evaluation results of Figure 8.7(b) indicate a strong leakage for consecutive round values although round outputs are randomized.

Profile 3: Register Precharge. An additional register stage and random initialization counteracts Hamming distance leakage because buffered data is not overwritten by consecutive
results but random values. The analysis results of this profile using 1 million power measurements can be seen in Figures 8.8(a), 8.8(b), 8.8(c), and 8.8(d). As expected, this countermeasure reduces the leakage related to the XOR between consecutive rounds’ output (cf. Figure 8.8(b)) but does not prevent the leakage associated to the S-box computation neither for the 64 models of the output bits nor for the 16 models of the output value.

8.4.3 Combination of Countermeasures
As a last step, we investigate all possible combinations of two or more countermeasures in order to find the best solution. This directly leads to the following four profiles:

Profile 4: Decomposition and Precharge In such a setting, precharging only prevents Hamming distance leakage, but the leakage of the S-box cannot be compensated. Figures
8.4 Practical Evaluation

Figure 8.7: Profile 2 (Boolean Masking) with 1 000 000 traces

(a) S-box output bits (64 models)

(b) XOR of round in and round out (64 models)

(c) Output value of S-box $S_0$ (16 models)

(d) Output Value of S-box $S_1$ (16 Models)

Figure 8.8: Profile 3 (Register Precharge) with 1 000 000 traces

(a) S-box output bits (64 models)

(b) XOR of round in and round out (64 models)

(c) Output value of S-box $S_0$ (16 models)

(d) Output Value of S-box $S_1$ (16 Models)

8.9(a), 8.9(b), 8.9(c), and 8.9(d) show the $t$-test results for all 144 models, but only for test group 2 the leakage could be reduced almost to the thresholds. For all other models still some leakage is detectable.

Profile 5: Masking and Precharge  The evaluation results of the combination of Boolean masking and register precharge are shown in Figures 8.10(a), 8.10(b), 8.10(c), and 8.10(d). This combination already minimizes the leakage but still, in particular in Figure 8.10(a), some leakage is detectable using 1 million traces.

Profile 6: Decomposition and Masking  The $t$-test results of the combination of S-box decomposition and Boolean masking can be seen in Figures 8.11(a), 8.11(b), 8.11(c), and 8.11(d). Apparently, this combination cannot compensate the problems of both countermeasures (leakage related to the XOR between consecutive rounds’ output) and is not sufficient to prevent first-order leakage.
Chapter 8 Dynamic Logic Reconfiguration

Profile 7: Decomposition, Masking and Precharge This last profile combines all our proposed countermeasures in order to benefit from their advantages. Figures 8.12(a), 8.12(b), 8.12(c), and 8.12(d) show the result of the corresponding evaluations. In this case, we even recorded 10 million power traces, but could not detect any first-order leakage.

8.5 Conclusion

In this chapter, we have presented a novel method to realize a first-order masking scheme based on the process of dynamic reconfiguration in modern FPGAs. Since the scheme uses precomputed masked S-boxes realized by Configurable Look-Up Tables, its security is not affected by the known issues of masked hardware, e.g., glitches. By using reconfiguration we applied a dynamic decomposition of the S-boxes that includes precharging of registers as well. We have practically examined all countermeasures and their combinations using the specific t-test.
leakage assessment methodology and we can report that even after recording 10 million power traces, we were not able to detect any first-order leakage if all countermeasures are active.

In short, we demonstrated an effective technique for FPGA-based platforms to achieve first-order SCA resistance. Compared to the known schemes, e.g., partial reconfiguration, our proposed solution has still reasonable overheads.
Chapter 9
Pitfalls of Memory Primitives

Block Memory Content Scrambling has been presented at CHES 2011 as an effective way of implementing first-order side-channel protection for cryptographic primitives. However, it comes at cost of a significantly increased reconfiguration time for the mask update. Based on joint work with Oliver Mischke, Amir Moradi, and Tim Güneysu [SMMG15a], this chapter analyzes alternative ways to implement dynamic first-order masking of AES with randomized look-up tables that can reduce this mask update time. In contrast to the previous chapter, where we considered CFGLUTs, in this chapter we extend the dynamic logic reconfiguration to various memory primitives including three Distributed Memory components (RAM32M, RAM64M, and RAM256X1S) and one BRAM primitive (RAMB8BWER). We provide a detailed study of the area and time overheads of each component and implementation technique with respect to the operation (encryption) as well as reconfiguration (mask update) phase. We further evaluate and compare the security of each technique based on practical measurements and consecutive leakage assessment.

Contents of this Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Introduction</td>
<td>103</td>
</tr>
<tr>
<td>9.2 Preliminaries</td>
<td>104</td>
</tr>
<tr>
<td>9.3 Design</td>
<td>108</td>
</tr>
<tr>
<td>9.4 Evaluation</td>
<td>110</td>
</tr>
<tr>
<td>9.5 Conclusion</td>
<td>112</td>
</tr>
</tbody>
</table>

9.1 Introduction

Randomizing look-up tables is a common countermeasure approach against SCA for software-based cryptographic implementations and effective in hardware to prevent information leakage due to glitches and early evaluation. A realization of such an approach called Block Memory Content Scrambling has been presented in [GM11] and was particularly designed for FPGAs in order to randomize and scramble the content of BRAMs. Since this approach notably suffers from slow performance due to a frequent and slow mask update and reconfiguration process, the previous chapter dealt with an alternative solution based on CFGLUTs in order to implement randomized look-up tables more efficiently. Based on build-in hardware structures of modern
FPGAs, the dynamic logic reconfiguration enables efficient cryptographic hardware agility for secure look-up table randomization.

In the course of this chapter, we will revise the concept of this dynamic logic reconfiguration approach by investigating different memory primitives. While the initial dynamic logic reconfiguration solely focuses on CFGLUTs and Shift Register LUTs, our novel approach evaluates different RAM-based interfaces to access the configuration memory within any Slice-M. More precisely, this evaluation analyzes and compares the performance and security of different solutions either based on Distributed Memory or BRAM.

9.1.1 Contribution

In this chapter, we analyze the suitability of different Xilinx FPGA memory primitives to prevent first-order side-channel leakage by masked look-up tables. Besides using larger dual-port BRAM primitives (as used in the original BMS publication [GM11]), it is also possible to use smaller single-port BRAMs as well as distributed RAM elements which are realized in Slice-M LUTs of modern Xilinx FPGAs [Xil13]. With the introduction of Xilinx’ Virtex-5 platform Slice-M have become capable to hold 256 bits of memory that is a perfect fit for an $8 \times 256$-bit AES S-box. In particular RAM32M, RAM64M, and RAM256X1S are the primitives which can be used to build a randomly permuted (masked) S-box. Although reconfiguration time becomes notably shorter for smaller RAM module sizes, the total area requirements of each masked S-box increases.

For evaluation, we apply the non-specific $t$-test as a general leakage assessment methodology [GJJR11] to analyze the SCA resistance of each scheme. We show that due to their intrinsic multi-LUT design, the distributed RAM elements still exhibit a first-order leakage so that they should not be used to implement masked designs. We conclude this chapter with presenting an efficient implementation of a small single-port BRAM-based design that achieves almost double the throughput of the original BMS scheme and still prevents first-order leakages.

9.2 Preliminaries

In this section we briefly describe the application of memory primitives in order to build randomized look-up tables to protect cipher implementations against first-order DPA attacks. Afterwards, we restate the concept of BMS initially introduced in [GM11].

9.2.1 Memory Primitives

Modern Xilinx FPGAs provide several memory primitives, e.g., Distributed Memory and general purpose BRAM, that can be used to build randomly permuted look-up tables. As presented in Chapter 2, Distributed Memory is enabled only at special Slices (Slice-M) by using the configuration registers within the LUTs as general purpose memory cells. Since this storage is usually constrained by the configuration size (between 16 and 64 bits), up to 4 LUTs of a single Slice-M can be combined in order to build larger RAMs. In addition, for designs requiring even larger amounts of memory, FPGAs provide general purpose BRAMs with sizes varying between 8kb and 32kb.
In the following we describe these memory primitives and their modes of operation individually and in detail, focusing on their application as a randomized look-up table (see [Xil13] for more information).

**RAM32M**: The **RAM32M** memory primitive is a multi-port RAM with synchronous write but asynchronous read capability implemented in Distributed Memory using the configuration memory of all four LUTs (including both outputs, i.e., O6 and O5) of a single Slice-M. It is organized as an 8-bit wide by 32 deep memory providing 4 individual read ports (each 2-bit wide) and a single write port (8-bit wide). If all read addresses are tied to the same value, this memory primitive becomes an $8 \times 32$ single-port RAM.

**RAM64M**: In contrast to the **RAM32M** primitive, the **RAM64M** module is a multi-port RAM with synchronous write and asynchronous read capability organized as 4-bit by 64 deep memory. This memory primitive also occupies four LUTs of a Slice-M but only uses the O6 outputs of each LUT. Similarly, if all 6-bit wide address ports are tied to the same value, this memory becomes a $4 \times 64$ single-port RAM.

**RAM256X1S**: Another option for Distributed Memory is **RAM256X1S**. This primitive is a single-port RAM with synchronous write and asynchronous read capability placed in a single Slice-M again using all four LUTs (combined by subsequent the MUXF7 and MUXF8 multiplexer instances). A **RAM256X1S** provides an 8-bit wide address port and a 1-bit wide read and write port and is organized as a $1 \times 256$ single-port RAM.

**RAMB8BWER**: The **RAMB8BWER** primitive is a true dual-port RAM with synchronous read and write capability. Instead of using configuration memory of special LUTs as Distributed Memory, this RAM instance uses a dedicated BRAM and offers 8kb data storage in addition to a 1kb parity memory. It is possible to define different options and widths for the read and write ports changing the memory configuration from $1 \times 8kb$ up to $9 \times 1kb$. The embedded input register causes this primitive to always require a clock cycle for a synchronous read. In addition, the output port can use an additional embedded register in order to buffer the memory output leading a total latency of two clock cycles for a read operation.
9.2.2 Randomized Look-Up Tables

Many symmetric ciphers use S-boxes, often represented by simple look-up tables, in order to include non-linearity into the encryption scheme. In FPGAs, this S-boxes can efficiently be realized either using LUTs (as well as distributed RAM) or BRAM depending on their size as well as the available resources.

SCA attacks target an intermediate value of a cipher, e.g., a part of the nonlinear layer. They predicted intermediate values, usually the input or output of a known S-box, in addition to a hypothetical power model contribute in a statistical analysis of e.g., power consumption traces in order to reveal the associated secret. In order to avoid side-channel leakages, hardware designers need to apply dedicated countermeasures e.g., masking. These countermeasures aim at randomizing intermediate values of a cipher implementation using uniformly-distributed random data (masks). In particular, the nonlinear layer in terms of look-up tables such as S-boxes (or T-Tables) has to be adapted depending on the taken random mask. Usually this is done by scrambling the S-box content based on an input mask \( m \) and adding an output mask \( n \) to the content (Boolean masking), so that the masked S-box \( S' \) is precomputed as:

\[
S'(x \oplus m) = S(x) \oplus n
\]

As mentioned before, look-up table based S-boxes can be implemented using Distributed Memory or BRAM. Due to their reconfiguration feature, the previously presented memory primitives can be employed to implement randomized look-up tables as well. The following Figures exemplarily present a part of the structure of an AES S-box using different memory primitives: Figure 9.1 (using RAM32M), Figure 9.2 (using RAM64M), Figure 9.3 (using RAM256X1S) and Figure 9.4 (using RAMB8BWER).

Each of the Distributed Memory designs presented in these Figures realizes one bit of the AES S-box. More precisely, each of them receives an 8-bit input \( S_{IN} \), and provides one output bit \( S_{OUT} \). Depending on their read and write port width, the configuration to update the look-up table is defined. For example, the content of 8 bits of a RAM32M can be updated in one clock.
9.2 Preliminaries

Figure 9.3: S-box architecture using \texttt{RAM256X1S}

cycle (Figure 9.1) while at most 4 bits of \texttt{RAM64M} and 1 bit of \texttt{RAM256X1S} can be simultaneously updated. This clearly affects the efficiency of the update (reconfiguration) process. Respectively, extra components, i.e., the multiplexers in Figure 9.1, have to be placed out of the Slice-M to build a $1 \times 256$ memory. With respect to this issue \texttt{RAM256X1S} is the most efficient one while the time required to update its content is considerably higher than the other distributed memory primitives.

9.2.3 Block Memory Content Scrambling

The main idea of BMS is to store two S-/T-Tables in parallel into a dual-port BRAM where one is called active context and the other one passive. While the active context is used for the encryption process via the first port of the BRAM, the passive context is scrambled by means of the second port. During the scrambling process, already masked data is read from the active context, and updated by a given fresh mask before it is written back to the passive context. After the encryption and memory content scrambling process have finished, the contexts are swapped i.e., the passive context becomes active and is used for the encryption process while the active context becomes passive and is updated using a new (random) mask. This scrambling scheme exploits the true dual-port capability of BRAMs in order to randomize look-up tables such as S-boxes or T-Tables without affecting the throughput of the encryption scheme. Despite many advantages, this scheme still comes with:

- **Area overhead, since it doubles the memory requirements because every look-up table has to be stored twice (active and passive).**

- **Additional latency for a mask update process, as the scrambling (updating) process needs 512 clock cycles. Hence it often happens that the consecutive encryptions share the masks since the scrambling process is not finished when the second plaintext is given.**
9.3 Design

This section briefly explains the underlying masking scheme of our AES implementation and its basic hardware architecture. Afterwards, different approaches using the Distributed Memory and the BRAM primitives are compared.

9.3.1 Masking Architecture

The architecture of our design of the AES-128 encryption function (for a Spartan-6 FPGA) is shown in Figure 9.5. We opted to implement an incremental and round-based architecture and derive the round keys on-the-fly. The data path has a width of 128 bits, and the SubBytes layer consists of 16 parallel reconfigurable S-boxes. ShiftRows and MixColumns (in parallel on all 4 columns) are applied jointly at one clock cycle.

In contrast to the originally proposed BMS scheme, our design follows an approach based on an update-prior-to-encryption fashion. Thus, before each encryption the randomized look-up tables are regenerated. During each encryption the masks stay constant. In other words, the same masks are used for all cipher rounds during one encryption. The initial plaintext is masked with \((m \oplus m')\) while all round keys are masked with \(m'\) (\(m\) and \(m'\) independent of each other and each 128-bit). Therefore, after the key addition the SubBytes input mask is \(m\) (see Figure 9.5). The randomized look-up tables (masked SubBytes) are configured with \(m\) as the input mask and \(SR^{-1}(MC^{-1}(m \oplus m'))\) as the output mask. Applying the ShiftRows and MixColumns operations transforms the mask again to \((m \oplus m')\) as the mask of the round output. Hence, after each cipher round the input to the next round is masked with \((m \oplus m')\) and no mask correction (see [BCL12, NSGD12]) is required. For the last round, the MixColumns operation is omitted and the returned ciphertext is masked with \(MC^{-1}(m \oplus m') \oplus m'\).

Reusing the masks for all cipher rounds has a known drawback if the round register consecutively stores the intermediate values with the same mask. In such a case, the leakage associated to the register update, e.g., a Hamming distance (HD) model, is easily extractable. If \(x \oplus m\) and \(y \oplus m\) are consecutively stored in a register,

\[
HD(x \oplus m, y \oplus m) = HW(x \oplus y)
\]
is independent of the mask. Hence, we avoid such an issue by surrounding each S-box with two register stages, one before and one after the SubBytes operation (see Figure 9.5). At power-up both registers are precharged with a zero value, and at only one clock cycle the input multiplexer passes the masked plaintext \((p \oplus m \oplus m')\). Since one of the register stages therefore holds some value depending on a random mask of a previous encryption, the correct encryption rounds are interleaved with random (dummy) operations.

Employing this technique leads to reduced throughput due to the prior look-up table update phase as well as the fact that each cipher round requires two clock cycles. However, compared to BMS [GM11] our design reduces the area overhead as well as the amount of required randomness to 256-bit per encryption \((m \text{ and } m')\). Further, this scheme is suitable for the Distributed Memory primitives as well as for the BRAM which allows a fair comparison. In case the BRAM is used, the registers (before and after the SubBytes) are removed. Instead, the input and output registers of the BRAM are employed as the two-stage state registers.

### 9.3.2 Comparison of S-box designs

Table 9.1 provides a comparison of area and time requirements of the randomized look-up tables using different memory primitives and the associated configuration logic and in Table 9.2 we give an overview of the resource requirements of the entire AES encryption as well as an estimation of the maximum frequency and throughput. Compared to the originally proposed BMS scheme, our masked design based on the BRAM (RAMBBW) halves the reconfiguration time, hence nearly doubling the maximum throughput. In case the Distributed Memory primitives are employed, the maximum frequency can even be increased except for the RAM32M due to its more complex reconfiguration circuit. Besides, the RAM32M leads to the highest throughput as its reconfiguration time is extremely shorter than the others. Note that in the reported performance figures we omitted the area required for the generation of the random masks.
Table 9.1: Comparison of S-boxes for different memory primitives

<table>
<thead>
<tr>
<th>Memory Primitive</th>
<th>Logic (LUT)</th>
<th>S-boxes Memory (LUTRAM)</th>
<th>Configuration Memory (BRAM)</th>
<th>Logic (LUT)</th>
<th>Memory (FF)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Block Memory</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRAM (BMS)</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>1706*</td>
<td>1169*</td>
</tr>
<tr>
<td>RAMBBBWER</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>298</td>
<td>8</td>
</tr>
<tr>
<td><strong>Distributed Memory</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAM256X1S</td>
<td>128</td>
<td>512</td>
<td>0</td>
<td>298</td>
<td>8</td>
</tr>
<tr>
<td>RAM64X1</td>
<td>768</td>
<td>512</td>
<td>0</td>
<td>727</td>
<td>6</td>
</tr>
<tr>
<td>RAM32X1</td>
<td>1920</td>
<td>512</td>
<td>0</td>
<td>1222</td>
<td>5</td>
</tr>
</tbody>
</table>

* These values are based on a Virtex-II Pro implementation and taken from [GM11]. For a Spartan-6 the resulting design would be slightly smaller.

9.4 Evaluation

We employed a SAKURA-G platform [GIS14], i.e., a Spartan-6 FPGA, for practical side-channel evaluations. The power consumption traces have been measured by means of a LeCroy WaveRunner HRO 66Zi oscilloscope with a 1 Ω resistor in the Vdd path capturing the embedded amplifier output of the SAKURA-G. We recorded the traces at a sampling rate of 1 GS/s and the bandwidth limit of 20 MHz while the design was running at a low clock frequency of 3 MHz to reduce the noise caused by the overlap of the power traces.

9.4.1 Results

In the following we present the results of the security evaluation concerning side-channel resistance of randomized look-up tables using the introduced memory primitives by applying the above-explained non-specific t-test. Since we identified four potential memory elements (see Section 9.2.1), the evaluations are grouped into four different profiles respectively.

Note that all our measurements cover only the time period related to the encryption, and we ignored to measure the power consumption when the reconfiguration of the look-up tables is in process (prior to each encryption). As explained in Section 9.3, we kept the design architecture of all profiles the same. Hence the power traces of other profiles look like the same, but for the design profile with RAM32X1 the traces show slightly higher peak-to-peak amplitude due to its more complex architecture regarding the extra multiplexers out of the RAM slices.

For each profile we collected at least 1 million traces for a non-specific t-test. During all the measurements fresh masks are randomly generated by means of an AES engine running in counter mode prior to each encryption, i.e., no mask is reused. The masked plaintext in addition to the corresponding masks are sent from the control FPGA to the target FPGA (SAKURA-G). After finishing the look-up table reconfiguration followed by the encryption process on the target FPGA, the masked ciphertext is sent back to the control FPGA, where it is unmasked for a consistency check.

**Profile 1: Small RAM (RAM32X1).** By means of this profile we evaluate the leakage of the randomized look-up table realized by RAM32X1 memory primitives. Although this variant has the highest resource consumption, it offers the best throughput. Figure 9.6(a) shows the
Table 9.2: Time and resource requirements of entire AES (encryption only)

<table>
<thead>
<tr>
<th>Memory Primitive</th>
<th>Logic</th>
<th>AES Encryption</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(LUT)</td>
<td>Memory</td>
<td>Reconfiguration</td>
</tr>
<tr>
<td>Block Memory</td>
<td></td>
<td></td>
<td>(Cycles)</td>
</tr>
<tr>
<td>BRAM (BMS)</td>
<td>2888</td>
<td>2351</td>
<td>0</td>
</tr>
<tr>
<td>RAMB8BWER</td>
<td>1284</td>
<td>415</td>
<td>0</td>
</tr>
<tr>
<td>Distributed Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAM256X1S</td>
<td>1284</td>
<td>543</td>
<td>512</td>
</tr>
<tr>
<td>RAM64K</td>
<td>2337</td>
<td>541</td>
<td>512</td>
</tr>
<tr>
<td>RAM32K</td>
<td>4000</td>
<td>540</td>
<td>512</td>
</tr>
</tbody>
</table>

* Reconfiguration can be done in parallel when reusing the mask for multiple encryptions without affecting the throughput. For a fair comparison we avoid the mask reuse in BMS as well.

result of the corresponding non-specific t-test using 1 million traces (i.e., about 500,000 traces of encrypting the fix plaintext and the rest for the random ones). Unexpectedly, the test exhibits first-order leakages. Indeed, the t statistics are much higher than the threshold, that confidently argue the vulnerability of the design.

Profile 2: Medium RAM (RAM64K). The result of the same test on the second profile, i.e., the one where the randomized look-up tables are implemented by RAM64K instances, is shown in Figure 9.6(b). We observed the same issue, i.e., unexpected first-order leakages. Interestingly, the amount of leakage is higher compared to that of Profile 1, although its S-box design is more compact.

Profile 3: Large RAM (RAM256X1S). The most compact and dense implementation for a randomized look-up table using Distributed Memory (i.e., the RAM256X1S memory primitives) on a Spartan-6 FPGA, places a complete single AES S-box and the subsequent registers into only 8 slices. However, the same as for the two former design profiles, first-order leakage is still detectable which can be seen in Figure 9.6(c).

Profile 4: Block RAM (RAMB8BWER). As the last profile, we evaluated the application of BRAMs instead of the Distributed Memory. Since each BRAM internally has a register stage for the input and an optional one for the output, by employing a RAMB8BWER instance for each S-box we used also both internal registers of the BRAM and avoided the external registers used in the other profiles (see Figure 9.5). Since we did not observe any first-order leakage using the same number of traces as used for the other profiles, we performed the evaluation using 10 million traces. The corresponding result is shown in Figure 9.6(d) indicating the ability of the design to prevent any first-order leakage.

In fact, the results we presented before infer the pitfall of using Distributed Memory (of FPGAs) to realize randomized (masked) look-up tables. While the internal architecture of such memory primitives is not completely clear to us, we are confident that the observed leakage is due to the internal multiplexers of such memory modules. We should highlight that the randomized look-up tables (in our designs) receive only the masked inputs and provide the masked outputs. Neither the input mask nor the output mask is given to the memory module. Further, the input masks and output masks are independent of each other. As a result – also
Chapter 9 Pitfalls of Memory Primitives

confirmed by the evaluation result of Profile 4 – the exhibited leakage is purely related to the internal architecture of the Distributed Memory modules.

9.5 Conclusion

In this chapter, we have given a comparative study on the suitability of Xilinx FPGA memory primitives to implement a side-channel countermeasure based on randomized (masked) look-up tables. We have shown that the use of Distributed Memory primitives like RAM32M, RAM64M, and RAM256X1S causes an otherwise secure scheme to exhibit first-order side-channel leakage. Such unexpected leakage is due the internal architecture of the Distributed Memory primitives (Slice-M). Since there is no other public document on the details of such modules except for [Xil13], we cannot localize the source of such leakage. When keeping the very same design but only replacing the distributed RAMs by small BRAMs to store the masked tables, no leakages were detected applying the general non-specific leakage assessment methodology on 10 million captured power traces.

Our design solution using BRAM (RAMB8BWER) achieves almost double the throughput compared to the original BMS mainly because of the reduced reconfiguration time of the masked S-boxes. It also requires less randomness. The BMS scheme is a T-table implementation which requires $16 \times 32$ random bits to mask the T-tables output while we only require $2 \times 128$ bits of randomness. The reason for this difference is that we are only implementing the $2^8 \times 8$-bit AES S-box as masked tables (compared to $2^8 \times 32$-bit T-tables) while the other parts (all linear) of the encryption are implemented by combinatorial logic.
Chapter 10

Affine Equivalence and its Applications

In this chapter, we explore the possibilities of affine equivalences of Boolean functions in order to thwart higher-order attacks on first-order TIs. In particular, instead of increasing the order of resistance by employing higher-order TIs, we go toward introducing structured randomness into the implementation itself. In that sense, our construction, based on joint work with Amir Moradi and Tim Güneysu and published results in [SMG15a], is a combination of masking and hiding, which is dedicated to TI designs and pursues the approach of constant, algorithmic restructuring of hardware implementations. Eventually, we can prove that such a combination hardens a design practically against higher-order attacks such that these attacks cannot be successfully mounted. We show that the area overhead of our construction is paid off by its ability to avoid higher-order leakages to be practically exploitable.

Contents of this Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 Introduction</td>
<td>113</td>
</tr>
<tr>
<td>10.2 Background</td>
<td>114</td>
</tr>
<tr>
<td>10.3 Design Considerations</td>
<td>116</td>
</tr>
<tr>
<td>10.4 Evaluation</td>
<td>121</td>
</tr>
<tr>
<td>10.5 Discussions</td>
<td>124</td>
</tr>
<tr>
<td>10.6 Conclusion</td>
<td>126</td>
</tr>
</tbody>
</table>

10.1 Introduction

SCA exploits information leakage related to cryptographic device internals e.g., by analyzing the power consumption [KJJ99]. Hence, integration of dedicated countermeasures to SCA attacks into security-sensitive applications is essential particularly in case of pervasive applications (see [EKM+08, RRST02, ZYSQ13]). Among the known countermeasures, masking as a form of secret sharing scheme has been extensively studied by the academic communities [CJRR99, MOP07]. Based on Boolean masking and multi-party computation concept, TI has been developed particularly for hardware platforms [NRS11]. Since the TI concept is initially bases on counteracting only first-order attacks, trivially higher-order analysis, which makes use of statistical moments at higher order to exploit the leakages, can still recover the secrets. Hence, the TI has been
extended to higher orders [BGN+14b] which might be limited to univariate settings [RBN+15]. In addition to its area and time overheads, which increase with the desired security order, the minimum number of shares also naturally increases, e.g., 3 shares for the first-order, 5 shares for the second-order, and at least 7 shares for the third-order security.

10.1.1 Contribution

In this chapter, we look at the feasibility of higher-order attacks on first-order secure TI designs from another perspective. Instead of increasing the resistance against higher-order attacks by employing higher-order TIs, we intend to introduce structured randomness into a first-order secure TI. Our goal is to practically harden designs against higher-order attacks that are known to be sensitive to noise.

Concretely, we investigate the PRESENT [BKL+07] S-box under first-order secure TI settings that is decomposed into two quadratic functions thereby allowing the minimum number of three shares. By changing the decompositions during each operation of the device, we can introduce (extra) randomness to the implementation itself. In particular, we present different approaches to find and generate these decompositions on an FPGA platform and compare them in terms of area and time overheads. More importantly, we examine and compare the practical evaluation results of our constructions using a state-of-the-art leakage assessment methodology [GJJR11] at higher statistical orders.

Our proposed approach, which can be considered as a hiding technique, is combined with a first-order TI which provides provably security and first-order resistance. Therefore, although such a combination leads to higher area overhead, it brings its own advantage, i.e., practically avoiding the feasibility of higher-order attacks.

10.2 Background

In the following, we briefly discuss the concept of S-box decomposition for the particular case of the PRESENT S-box and its decomposition into two quadratic functions before we explain the notion of affine equivalences for (vectorial) Boolean functions.

10.2.1 S-Box Decomposition

Since the nonlinear part of most block ciphers, i.e., the S-box, has algebraic degree of \( t > 2 \), the number of input and output shares is \( n, m > 3 \), which directly affects the hardware circuit complexity and its area overhead. Therefore, it is preferable to decompose the S-box \( S \) into smaller functions, e.g., \( g \circ f \), each of them with maximum algebraic degree of 2. It is noteworthy that if \( S \) is a bijection, each of the smaller functions (here in this case \( g \) and \( f \)) also have to be a bijection. Such a trick helps keeping the number of shares for input and output at minimum, i.e., \( n = m = 3 \). However, it comes with the disadvantage of the necessity to place a register between each two consecutive TI smaller functions to avoid the glitches being propagated.

Although such a composition is feasible in case of small S-boxes (at least up to 6-bit permutations [BNN+15]), it is still challenging to find such decompositions for \( 8 \times 8 \) S-boxes. As stated before, the target of this chapter is an implementation of PRESENT cipher, which involves a \( 4 \times 4 \) invertible cubic S-box (i.e., with the algebraic degree of 3) with Truth table
10.2 Background

Therefore, all the representations below are coordinated based on 4-bit bijections.

The PRESENT S-box. In [PMK+11], where the first TI of PRESENT is presented, the authors gave a decomposition of the PRESENT S-box by two quadratic functions, i.e., each of them with the algebraic degree of 2. Later the authors of [BNN+12] and [BNN+15] presented a systematic approach which allows deriving the TI of all 4-bit bijections. In their seminal work they provided 302 classes of 4-bit bijections, with the application that every 4-bit bijection is affine equivalent to only one of such 302 classes. Based on their classification, the PRESENT S-box belongs to the cubic class $C_{266}^4$ with Truth Table 0123468A5BCFED97. It other words, it is possible to write the PRESENT S-box as $S : A' \circ C_{266}^4 \circ A$, where $A'$ and $A$ are 4-bit bijective affine functions. Therefore, given the uniform TI representation of $C_{266}^4$ one can easily apply $A$ on all input shares and $A'$ on all output shares to obtain a uniform TI of the PRESENT S-box.

Decomposition into Quadratic Functions. Further, as stated in [BNN+15], the function $C_{266}^4$ can be decomposed into two 4-bit quadratic bijections belonging to the following seven combinations of classes:

$$
(Q_{12} \circ Q_{12}), (Q_{293} \circ Q_{300}), (Q_{294} \circ Q_{299}), (Q_{299} \circ Q_{294}), (Q_{299} \circ Q_{299}), (Q_{300} \circ Q_{293}), \text{ and } (Q_{300} \circ Q_{300}).
$$

However, the uniform TI of the quadratic class $Q_{300}$ with 3 shares can only be achieved if it is again decomposed in two parts. Therefore, the above decompositions in which $Q_{300}$ is involved need to be implemented in 3 stages if the minimum number of 3 shares is desired. Excluding such decompositions we have four options to decompose the PRESENT S-box in two stages with 3-share uniform TI since the PRESENT S-box is affine equivalent to $C_{266}^4$.

For the sake of simplicity – as an example – we consider the first decomposition, i.e., $Q_{12} \circ Q_{12}$, which indicates that it is possible to write the PRESENT S-box as $S : A'' \circ Q_{12} \circ A' \circ Q_{12} \circ A$, where all three $A''$, $A'$, and $A$ are 4-bit affine bijections. Thanks to the classifications given in [BNN+15]
a uniform first-order TI of $Q_{12}$ can be achieved by direct sharing. For $Q_{12}:0123456789CDEFAB$ we can write:

$$
\begin{align*}
\langle x_0, x_1, x_2, x_3 \rangle & \rightarrow \langle y_0, y_1, y_2, y_3 \rangle \\
\langle x_0, x_1, x_2, x_3 \rangle & \rightarrow \langle y_0, y_1, y_2, y_3 \rangle \\
\langle x_0, x_1, x_2, x_3 \rangle & \rightarrow \langle y_0, y_1, y_2, y_3 \rangle
\end{align*}
$$

with $\langle x_0, x_1, x_2, x_3 \rangle$ the 4-bit input, $\langle y_0, y_1, y_2, y_3 \rangle$ the 4-bit output, and $x_0$ and $y_0$ the least significant bits.

Consequently, the component functions of the uniform first-order TI of $Q_{12}$ can be derived by $f_{Q_{12}}(\langle x_0, x_1, x_2, x_3 \rangle, \langle x_0, x_1, x_2, x_3 \rangle) = \langle y_0, y_1, y_2, y_3 \rangle$ as

$$
\begin{align*}
y_0 &= x_0, \\
y_1 &= x_1 + x_1 x_3 + x_2 x_3, \\
y_2 &= x_2 + x_1 x_3, \\
y_3 &= x_3.
\end{align*}
$$

The three 4-bit output shares provided by $f_{Q_{12}}^2$, $f_{Q_{12}}^3$, and $f_{Q_{12}}^4$ make a uniform first-order TI of $Q_{12}$. Since the affine transformations $(A, A', A'')$ do not change the uniformity, by applying them on each 4-bit share separately we can construct a 3-share uniform first-order TI of the PRESENT S-box. Figure 10.1 shows the graphical view of such a construction, and the detailed formulas of the component functions are given in Appendix 14.

10.2.2 Affine Equivalence

In order to find such affine functions we give a pseudo code in Algorithm 2 which is mainly formed following [BCBP03]. The algorithm is based on precomputation of all $4 \times 4$ linear functions, i.e., 20 160 cases, each of which is represented by a $4 \times 4$ binary matrix with columns $(c_0, c_1, c_2, c_3)$. Hence, each affine function $A$ is considered as a matrix multiplication followed by a constant addition $A(x) = [c_0 \ c_1 \ c_2 \ c_3] \cdot x \oplus c$.

Given the PRESENT S-box and $f = g = Q_{12}$ the algorithm finds 147 456 such 3-tuple affine bijections $(A, A', A'')$. Table 10.1 lists the number of found affine triples for each of the aforementioned decompositions.

10.3 Design Considerations

This section briefly demonstrates the architecture the PRESENT TI which we have implemented. Afterwards, different approaches for generating and exchanging affine triples are presented and compared.

10.3.1 Threshold Implementation of PRESENT

Figure 10.2 gives an overview of our hardware architecture implemented on an Xilinx Spartan-6 FPGA. We opted to implement the PRESENT encryption scheme in a round-based manner.
Algorithm 2: Find affine equivalent triples

\begin{algorithm}
\textbf{Input} : $\mathcal{L}^4$: all $4 \times 4$ linear permutations, $S$: targeted S-box, $F$, $G$: targeted functions
\textbf{Output}: $\mathcal{A}$: all $(A, A', A'')$ as $S : A'' \circ G \circ A' \circ F \circ A$

$\mathcal{A} \leftarrow \emptyset$
\For{$\forall L \in \mathcal{L}^4$, $\forall c \in \{0, 1\}^4$} {
\For{$\forall L' \in \mathcal{L}^4$, $\forall c' \in \{0, 1\}^4$} {
form affine $A$ by $L$ and constant $c$
form affine $A'$ by $L'$ and constant $c'$
\begin{align*}
c'' & \leftarrow G \left(A' \left(F \left(A \left(S^{-1}(0)\right)\right)\right)\right) \\
c_1'' & \leftarrow G \left(A' \left(F \left(A \left(S^{-1}(1)\right)\right)\right)\right) \oplus c'' \\
c_2'' & \leftarrow G \left(A' \left(F \left(A \left(S^{-1}(2)\right)\right)\right)\right) \oplus c'' \\
c_3'' & \leftarrow G \left(A' \left(F \left(A \left(S^{-1}(4)\right)\right)\right)\right) \oplus c'' \\
c_4'' & \leftarrow G \left(A' \left(F \left(A \left(S^{-1}(8)\right)\right)\right)\right) \oplus c''
\end{align*}
form affine $A''^{-1}$ by columns $(c_1'', c_2'', c_3'', c_4'')$ and constant $c''$
\If {$\forall y \in \{0, 1\}^4 \setminus \{0, 1, 2, 4, 8\}$, $G \left(A' \left(F \left(A \left(S^{-1}(y)\right)\right)\right)\right) \neq A''^{-1}(y)$} {
\textbf{derive} affine $A'$ as the inverse of $A''^{-1}$
$\mathcal{A} \leftarrow \mathcal{A} \cup \{(A, A', A'')\}$
}
}
end
end
end
\end{algorithm}

along with the 128-bit key schedule variant where the sub-keys are derived on-the-fly. The substitution layer uses the first-order TI of the PRESENT S-box shown in Figure 10.1 and implements 16 S-boxes in parallel before the permutation is applied bit-wise to all 64-bit states individually. Due to the additional register stage within the TI S-box each round requires two clock cycles.

As stated in Section 10.2.2, given a certain decomposition there exist many triple affine functions to realize a uniform first-order TI of the PRESENT S-box. Our goal is to randomly change such affine functions on-the-fly, that it first does not affect the correct functionality of the S-box, and second randomizes the intermediate values – particularly the shared $Q_{12}$ inputs – with the aim of hardening our construction against higher-order attacks. As shown in Figure 10.2 all S-boxes share the same affine triple. In other words, at the start of each encryption an affine triple is randomly selected, and all S-boxes are configured accordingly. Although it is possible to change the affine triples more frequently, we kept the selected them for an entire encryption process. To this end, we need an architecture capable to derive and change the affine triples randomly. In the following, we discuss about different approaches to realize and implement such a part of the design.
### 10.3.2 Searching for the Affine Triples

In a first attempt, we decided to implement Algorithm 2 as a hardware circuit which searches for the affine triples in parallel to the encryption module. Each affine triple that is found is stored into a FIFO memory, and prior to each encryption one affine triple is taken from the FIFO in order to configure the corresponding parts of the TI S-boxes. In case the FIFO is empty, the current configuration stays unchanged and previous affine triple is used again. Due to the fact that our proposed search algorithm is not time-invariant, i.e., new affine triples are not found periodically, some results are used multiple times in a row while others are only used once. Since the efficiency of most SCA countermeasures depends on the uniformity of used randomness, such an implementation may not fully achieve the desired goal (i.e., hardening against higher-order attacks) if certain affine triples are used more often that the others.

Hence, one solution to increase the outcome and find affine triples more frequently, is to run the search circuit with a higher clock frequency compared to that of the encryption circuit. Although this measure is limited, it at least alleviates the problem of changing S-boxes not periodically. On the other hand, if affine triples are found too fast this may cause a FIFO overflow. In this case either some results should be ignored or the search circuit has to stop which again requires some overhead in terms of additional control logic.

### 10.3.3 Selecting Precomputed Affine Triples

As stated in Table 10.1, considering the decomposition $Q_{12} \circ Q_{12}$, there exist 147,456 triple affines $(A, A', A'')$. Each single affine transformation is a 4-bit permutation, and it can be represented as a look-up table containing sixteen 4-bit entries which then maps to 64 bits of memory. In total, storing all possible affine triples results in about 27 MBit memory. However, the employed Xilinx Spartan-6 FPGA (LX75) offers only 3 MBit storage in terms of general purpose BRAM. Therefore, alternative approaches to generate the affine equivalent triples are necessary.

Instead of storing the affine triples in a look-up table, in our second approach we represent an exemplary affine as $A(x) = L \cdot x \oplus c$, with $x$ as a 4-bit vector, $L$ a $4 \times 4$ binary matrix and $c$ a 4-bit constant. In this case, only the binary matrix and the constant need to be stored which reduces the memory requirements to 20 bits per affine. However, still more than 8 Mbit memory are necessary to store all affine triples. Therefore, we could store only a fraction of all possible affine triples. However, assuming that a higher number of S-box alternatives due to changing the affine equivalent triples, will improve the resistance of our design against higher-order side-channel attacks, again this approach might not harden our design to its full extent. Still, as an advantage in comparison to the searching, this approach allows an immediate, time-invariant and uniformly random changing of the precomputed (subset) of affine triples. As an example,
we instantiate this option with 16,384 affine triples that occupy 60 BRAMs of the Spartan-6 (LX75) FPGA.

### 10.3.4 Generating Affine Triples On-the-fly

Eventually, a detailed analysis of the affine triples and their structure led to interesting observations that we discuss in the following.

**Observation 10.1.** The number of affine triples depends on the components of an underlying decomposition. For instance, in case of $Q_{299} \circ Q_{299}$, $448 \times 448$ triples exist and in case of $Q_{299} \circ Q_{294}$, $448 \times 512$ affine triples are valid solutions (see Table 10.1).

**Observation 10.2.** The total amount of affine triples is limited by the number of unique input affine functions $A$ and the number of output affine functions $A''$ such that $|A| \times |A''|$ gives the number of corresponding affine triples for a chosen decomposition.

In conclusion, this means that all affine triples of a decomposition can be generated by combining all input functions $A$ with all output transformations $A''$. Furthermore, we could observe that all affine functions $A$ (for each possible decomposition) consist of only a few linear matrices combined with certain constants. In particular, for instance in case of the decomposition $Q_{12} \circ Q_{12}$ all 384 input affine transformations $A$ can be generated by only 48 binary matrices $L$ each of which combined with 8 different constants $c \in \{0, \ldots, 7\}$ or $c \in \{8, \ldots, 15\}$. Indeed, the same holds for the 384 output affine functions $A''$ which again are defined by 48 binary matrices $L''$ and constants $c \in \{0, 1, 4, 5, 10, 11, 14, 15\}$ or $c \in \{2, 3, 6, 7, 8, 9, 12, 13\}$.

Given this information, it is sufficient to store only all relevant binary matrices $L$ and $L''$ in addition to a single bit indicating to which group their constants belong to. Hence, in total $48 \times 2 \times (16 + 1) = 1632$ bits of memory (fitting into a single BRAM) are required to store
all necessary data. Even better, by arranging the binary matrices in the memory smartly each group of corresponding constants can be derived from the address where the binary matrix is stored. Given two input and output affine transformations $A$ and $A''$, we need to derive the middle affine $A'$. To this end, an approach similar to Algorithm 2 can be used. If we represent the middle affine as $A'(x) = L' \cdot x \oplus c'$, the constant $c$ and the columns $(c'_1, c'_2, c'_3, c'_4)$ of the binary matrix $L$ can be derived as

$$c' = Q_{12}^{-1} \left(A''^{-1} \left(S \left(A^{-1} \left(Q_{12}^{-1} (0) \right) \right) \right) \right) \quad (10.3)$$

$$c'_1 = Q_{12}^{-1} \left(A''^{-1} \left(S \left(A^{-1} \left(Q_{12}^{-1} (1) \right) \right) \right) \oplus c' \quad (10.4)$$

$$c'_2 = Q_{12}^{-1} \left(A''^{-1} \left(S \left(A^{-1} \left(Q_{12}^{-1} (2) \right) \right) \right) \oplus c' \quad (10.5)$$

$$c'_3 = Q_{12}^{-1} \left(A''^{-1} \left(S \left(A^{-1} \left(Q_{12}^{-1} (4) \right) \right) \right) \oplus c' \quad (10.6)$$

$$c'_4 = Q_{12}^{-1} \left(A''^{-1} \left(S \left(A^{-1} \left(Q_{12}^{-1} (8) \right) \right) \right) \oplus c' \quad (10.7)$$

Obviously, this requires the inverse of both $A$ and $A''$. Since it is not efficient to derive such inverse affine functions on-the-fly, we need to store all binary matrices $L^{-1}$ and $L''^{-1}$ in addition to all $L$ and $L''$. Fortunately, all such binary matrices (requiring 3 kBits) still fit into a single 16-kBit BRAM of Spartan-6 FPGA. It is noteworthy that the constant of each inverse affine can be computed by $L^{-1} \cdot c$.

In summary, at the start of each encryption two $L$ and $L''$ (each of which from a set of 48 cases) are randomly selected, that needs $6 + 6$ bits of randomness. In addition, $3 + 3$ random bits are also required to form constants $c$ and $c''$. As exampled before, one bit of each constant should be additionally saved or derived from the address of the binary matrix. Therefore – excluding the masks required to represent the plaintext in a 3-share form for the TI design – in total 18 bits randomness is required for each encryption. For ASIC platforms, where block memories are not easily available, an alternative is to derive the content of binary matrices $L$ and $L''$ as Boolean functions over the given random bits. Hence, a fully combinatorial circuit can provide the input and output affine transformations followed (as before) by a module which retrieves the middle affine function.

### 10.3.5 Comparison of the Approaches

Table 10.2 gives an overview of the design for all three above-mentioned approaches to derive the affine triples. The table reports area overhead, reconfiguration time, and coverage of the affine triples’ space. Comparing the first, naive approach (of searching the affine triples in parallel to the encryption) to the approach of precomputing affine triples, the logic requirements could be dramatically decreased at cost of additional memory. However, at the same time, the amount of affine triples that are covered is limited potentially reducing the security gain. We should note that the 20 BRAMs used in our Search approach are due to the space required to store all $4 \times 4$ linear permutations $L^4$ necessary to run Algorithm 2 (excluding those required for the FIFO).

---

1For each selection $\in \{1, \ldots, 48\}$ reject sampling with 6-bit random should be used.
Table 10.2: Area and time overhead of different design approaches

<table>
<thead>
<tr>
<th>Section/Method/Module</th>
<th>Resource Utilization</th>
<th>Performance</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Logic (LUT)</td>
<td>Memory (FF)</td>
<td>Reconfiguration (Cycles)</td>
</tr>
<tr>
<td>10.3.2/Search</td>
<td>562</td>
<td>250</td>
<td>20</td>
</tr>
<tr>
<td>10.3.3/Precompute</td>
<td>204</td>
<td>0</td>
<td>60</td>
</tr>
<tr>
<td>10.3.4/Generate</td>
<td>114</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>Encryption [this work]</td>
<td>1720</td>
<td>722</td>
<td>0</td>
</tr>
<tr>
<td>[PMK+11]</td>
<td>641</td>
<td>384</td>
<td>0</td>
</tr>
<tr>
<td>[MW15]</td>
<td>808</td>
<td>384</td>
<td>0</td>
</tr>
<tr>
<td>[MW15]</td>
<td>2245</td>
<td>1680</td>
<td>0</td>
</tr>
</tbody>
</table>

Our last approach, where the affine triples are generated on-the-fly, seems to be the best choice. It not only leads to the least area overhead (both in terms of logic and memory requirements) but also covers the entire number of possible affine triples. We should note that this design needs a single clock cycle to derive the middle affine $A'$. Indeed the 114 LUTs (reported in Table 10.2) are mainly due to realization of Equation s (10.3)-(10.7) in a fully combinatorial fashion. Further, with respect to the design architecture of the encryption function (Figure 10.2), the quadratic component functions of $Q_{12}$ are implemented by LUTs, and the affine functions by fully combinatorial circuits realizing the binary matrix multiplication (AND operations) and XOR with the constant. Therefore, given 20 bits as the representation of the binary matrix and the constant, the circuit does not need any extra clock cycles for configuration.

Besides, Table 10.2 gives an overview of the area and speed overhead of our design compared to similar designs. For the first reference, the TI S-box is implemented by the design of [PMK+11] (i.e., without any random affine). The second reference implements both a first-order and a second-order TI S-box for PRESENT in a similar fashion (using $Q\_{294}$ and $Q\_{299}$ instead of $Q\_{12}$) but with fixed affine transformations. However, all numbers for the encryption function exclude the PRNG as well as the circuit which finds or derives the affine triples. Due to the extra logic to support and replace arbitrary affine triples, our design is certainly larger and slower.

### 10.4 Evaluation

We employed a SAKURA-G platform [Lab] equipped with a Spartan-6 FPGA for practical side-channel evaluations using the power consumption of the device. The power consumption traces have been measured and recorded by means of a digital oscilloscope with a 1 Ω resistor in the $V_{dd}$ path and capturing at the embedded amplifier of the SAKURA-G board. We sampled the voltage drop at a rate of 500 MS/s and a bandwidth limit of 20 MHz while the design was running at a low clock frequency of 3 MHz to reduce the noise caused by overlapping of the power traces.
In this section we present the result of our side-channel evaluations concerning the efficiency of our introduced approaches to avoid higher-order leakages. In order to solely evaluate the influence of randomly exchanging the affine triples we considered a single architecture in our evaluations. For a reference setup, the design is kept running with a constant affine triple\(^2\), and its evaluation results are compared to the case where the affine triples are randomly changed prior to each encryption. Note that in both cases (constant and random affine triples) the PRNG which provides masks for the initial first-order masking (with three shares) is kept active. In other words, both designs – based on the TI concept – are expected to provide first-order resistance, and their difference should be in exhibiting higher-order leakages.

In Section 10.3 we introduced three different approaches to derive affine triples. In the following, we provide practical evaluation results for all different options and highlight the issues and limitations of both first approaches (i.e., Search and Precompute). In summary, only the third approach presented in Section 10.3.4, i.e., generating affine triples on-the-fly, covers the entire possible search space and provides an adequate level of protection.

**Search.** The first approach followed the idea of exhaustively searching for the affine triples in a dedicated hardware circuit. This design results in a very area efficient architecture ensuring that all possible affine equivalent triples are found and used by the encryption core. In order

\(^2\)This has been easily done by fixing the corresponding 18 random bits.

Figure 10.3: Non-specific t-test results: Search
to accelerate the search, we increased the frequency of the searching circuit up to 96 MHz while still running the encryption at 3 MHz. Figure 10.3 provides the first-, second-, and third-order $t$-test results while using constant and random affine equivalent triples respectively. As expected, both investigations show a first-order resistance due to the implementation of a first-order TI. Considering the second-order vulnerability of the S-box with a constant affine equivalent triple we could detect some second-order leakage after using ten million power traces. When changing the affine equivalent triple randomly each time a new set of triples is found by the search circuit, the second-order leakage is still detectable using 10 million traces. The reasons for these observations are mainly due to the problems of this design already described in Section 10.3.2, i.e., the non-constant rate of finding affine equivalent triples.

**Precompute.** Since the second approach of storing some precomputed affine triples and the last approach of generating all affine triples on-the-fly mainly differ in terms of the number of affine equivalent triples that are covered, we omitted to discuss the design with precomputed triples and only provide the evaluation results in Figure 10.4. Eventually, we focused our analysis and discussion on the design which generates the triples on-the-fly (and therefore covers all possible affine equivalent triples).

**Generate.** Figure 10.5 shows two sample traces corresponding to the cases where the affine triples are either constant or random. The main difference between these two traces can be seen by a large power peak at the beginning of the trace belonging to the selection and computation.
of the random affine triples. In detail, such a peak indicates the corresponding clock cycle where the random affine is selected and the middle affine is computed (as stated in Section 10.3.5, it is implemented by a fully combinatorial circuit). The first-order, second-order and third-order \(t\)-test results are shown in Figure 10.6 respectively for both constant and random affine. As expected, both designs do not exhibit any first-order leakage confirming the validity of our setup and designs. However, changing the affine triples randomly could even avoid the second- and third-order leakage from being detectable. We should highlight that the evaluations of the design with a constant affine have been performed by 50 million traces while we continued the measurements and evaluations of the design with random affine triples up to 200 million traces.

10.5 Discussions

The scheme, which we have introduced here to harden higher-order attacks, at the first glance seems to just add more randomness to the design. We should stress that our approach is not the same as the concept of re-masking applied in [BNN+15, BGN+14a, MPL+11]. Re-masking (or mask refreshing) can be done e.g., by adding two new fresh random masks \(r^1\) and \(r^2\) to the input of a TI S-box in Figure 10.1 as \((x^1 \oplus r^1, x^2 \oplus r^2, x^3 \oplus r^1 \oplus r^2)\). Since our construction of the PRESENT TI S-box already provides uniformity, such a re-masking does not have any effect on the practical security of the design as both \((x^1, x^2, x^3)\) and \((x^1 \oplus r^1, x^2 \oplus r^2, x^3 \oplus r^1 \oplus r^2)\) are 3-share representations of \(x\). In contrast, in our approach e.g., the input affine \(A\) randomly changes. Hence, the input of the first \(Q_{12}\) function is a 3-share representation of \(A(x)\). Considering a certain \(x\), random selection of the input affine leads to random \(A(x)\) which is also represented by three Boolean shares. Therefore, the intermediate values of the S-box (at both stages) are not only randomized but also uniformly shared. As a result, hardening both second- and third-order attacks which make use of the leakage of the S-box can be justified. Note that since the S-box output stays valid as a Boolean shared representation of \(S(x)\) and random affine triples do not affect the permutation layer (of the PRESENT cipher), the key addition and the values stored in the state register, our approach is not expected to harden third-order attacks that target the leakage of these modules. However, our construction (which is a combination of masking and hiding) allows to achieve the presented efficiencies with low number of (extra) required randomness, i.e., 18 bits per encryption. Indeed, our approach might be seen as a form of shuffling which can be applied on the order of S-box executions in a serialized architecture. However, our construction is an algorithmic approach.
that is independent of the underlying architecture (e.g., serialized versus round-based) and allows hiding the exploitable higher-order leakages in a structured and systematic way.
10.6 Conclusion

In this chapter, we have explored the possibilities of affine equivalences of Boolean functions in order to introduce structured randomness into a hardware implementation and improve the resistance of first-order secure TIs to higher-order statistical moments. More precisely, we presented different approaches to find, precompute, and generate affine equivalent representations of the PRESENT S-box and used them to provide a hardware-agile cryptographic implementation based algorithmic reconfiguration. Eventually, we provide practical evaluation results based on a non-specific Welch’s $t$-test that confirm the practical resistance of our proposed strategy against higher-order side-channel analysis. In particular our last approach presents a very efficient alternative to HO-TI since it provides resistance against higher-order attacks (at least up to the third order) even after analyzing 200 million captured power traces. To this end, we can conclude that algorithmic reconfiguration along with first-order secure masking (i.e., TI) can provide higher-order security (at least from a practical point of view).
Chapter 11
White-Box Cryptography in the Gray Box

For an adversary which has full control over the execution process and the entire environment of a cryptographic implementation on an embedded device, standard countermeasures against physical attacks do not create an obstacle to access secret information. At this point, White-Box Cryptography is based on the fundamental principle of mapping an entire cryptographic architecture, in particular including the secret key, to a number of encoded look-up tables that shall even resist against inspection and decomposition of a white-box attacker. Adapting this concept to a gray-box scenario should hide sensitive information and implementation details from the attacker and could be used as a promising mitigation strategy to prevent side-channel leakage. In this chapter, we present a first white-box implementation of AES on reconfigurable hardware based on results provided in [SMG16b] (a joint work with Amir Moradi and Tim Güneysu) and evaluate our architecture under the assumption of a gray-box adversary. Unfortunately, we show that such an implementation, although originally developed for an even stronger white-box scenario, does not provide sufficient protection against SCA. In particular, static structure and behavior of the white-box implementation provide a point of leverage to circumvent the protection mechanisms. Hence, in a thorough mathematical analysis, we can identify the exact source of the observed leakage in order to present additional results which can be helpful to build stronger white-box cryptographic implementations.

Contents of this Chapter

<table>
<thead>
<tr>
<th>11.1 Introduction</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.2 Background</td>
<td>129</td>
</tr>
<tr>
<td>11.3 FPGA Implemtn</td>
<td>133</td>
</tr>
<tr>
<td>11.4 Side-Channel Analysis</td>
<td>135</td>
</tr>
<tr>
<td>11.5 Conclusion</td>
<td>141</td>
</tr>
</tbody>
</table>

11.1 Introduction
Initially the field of white-box cryptography was mainly motivated by applications of the field of Digital Rights Management (DRM) that aims to protect a secret key in a white-box environment, where an adversary has full control over the execution process and the environment of a cryptographic implementation. However, with the widespread emerging of embedded and
pervasive computing devices implementing cryptographic functions and primitives, the threat of white-box adversaries is no longer limited to cryptographic software implementations. Although, an adversary might be limited by the gray-box model in practice (i.e., he cannot control the execution process and the environment entirely), SCA attacks are well-known to be used to exploit information leakage related to the device internals e.g., by analyzing power consumption or electromagnetic radiations. Still, for successfully mounting such physical attacks, the attacker requires at least some knowledge about the internals in order to build adequate hypotheses that can be used, for example, for key extraction. In this context the nature of white-box cryptography that effectively disguises all internals and the secret key from the attacker by encoding them into tables, seems to yield some inherent resistance against such physical attacks.

11.1.1 Contribution
In this chapter, we propose a white-box implementation of AES dedicated to reconfigurable hardware. Although the white-box implementation of Chow et al. initially was proposed for software implementations, we show that the implementation can be mapped to existing reconfigurable hardware architectures. Note that only recent generations of reconfigurable hardware devices provide adequate amounts of resources to cope with the large memory requirements of white-box implementations. For this hardware implementation we next examine the vulnerability to SCA attacks assuming a gray-box adversary model. These results, obtained from an FPGA platform extend the observation by Bos et al. (in [BHMT16]). We show that SCA attacks such as classical DPA can reveal the secrets in hardware implementations applying white-box cryptography even in gray-box settings.

Finally, we perform a thorough mathematical investigation and analysis of the construction of look-up tables used in White-Box Cryptography. We explain and verify the reason behind the success of such (Differential Computational Analysis (DCA) and DPA) attacks what has not been addressed in the seminal work of Bos et al.. Our results give a better understanding of the mathematical foundations of these attacks which can pave the way for improved future white-box designs and implementations that are resistant against such analyses and threats.

11.1.2 Related Work
In 2002, first white-box implementations for DES [CEJvO02a] and AES [CEJvO02b] were proposed by Chow et al. in order to protect a secret key within a cryptographic implementation in presence of a white-box adversary. However, these seminal proposals and their implementations were soon shown to be vulnerable to differential cryptanalysis [GMQ07, WMGP07] as well as algebraic cryptanalytic attacks [BGE04, LRM+13, MGH08]. This led to some new proposals for white-box implementations of AES. In 2009, Xiao et al. in [XL09] proposed a variant of the design of Chow et al. using larger linear encodings, for which again a vulnerability against algebraic cryptanalytic attacks was identified in [MRP12]. Other approaches suggest to build white-box AES implementations using pertubations [BCD06] (which was broken in [MWP10]) or based on the concept of dual-ciphers [Kar10].

Recent work in [BCH14] aims to generalize and formalize notions for White-Box Cryptography and related attacks for any Substitution-Linear-Transformation (SLT) cipher presenting general attack strategies and upper bounds for their complexity. Besides the vulnerabilities against differential and algebraic cryptanalysis, Bos et al. in [BHMT16] showed that secret
keys of existing white-box implementations can be extracted by observing the addresses which are accessed during the execution if the external encodings are known to the adversary. The underlying so-called DCA applies the concept of DPA [KJJ99] on eavesdropped address bits.

A first white-box implementation in hardware has been proposed for the NOEKEON cipher in [BCD09] and [CFD+10] using 1-bit linear nibble encodings (i.e. masking with deterministic masks).

11.2 Background

This section introduces the basic concept of White-Box Cryptography and gives a detailed description of the seminal AES white-box implementation of Chow et al.

11.2.1 White-Box Cryptography

In general, the design strategy for white-box implementations of round-based symmetric block ciphers can be depicted as:

\[
\begin{align*}
(f(r+1))^{-1} \circ E_r \circ f^r \circ \cdots \circ (f(3))^{-1} \circ E^2 \circ f^2 \circ (f(2))^{-1} \circ E^1 \circ f^1 \\
\end{align*}
\]

where \(E_{i \in \{1, \ldots, r\}}\) is a single round instance of the block cipher and \(f^1\) respectively \((f^{r+1})^{-1}\) are considered as external input and output encoding of the white-box implementation (in order to prevent Code Lifting attacks [DLPR13]).

The white-box model has initially been proposed by Chow et al. [CEJvO02a] in 2002 when focusing on a fixed key implementation of the DES algorithm, and shortly afterwards a white-box implementation of the AES algorithm was presented [CEJvO02b]. In the following, we first introduce this seminal AES white-box implementation and discuss the design principles and known attacks and vulnerabilities under the white-box model before we show how to implement this design in hardware.

11.2.2 White-Box Implementation of AES

The architecture presented in [CEJvO02b] is a fixed key implementation with a fully unrolled design merging the atomic operations into a series of look-up tables. Basic design goals of this construction are to hide the key and algorithm structure through implementing the encryption as a network of randomized look-up tables. Each look-up table is encoded and protected individually using random linear and nonlinear bijections. Since a detailed discussion of the design would exceed the scope of this work we refer the interested reader to [Mui13] and restrict the discussion of the white-box implementation to its basic design principle and construction.
Design and Construction

The transformation of an unprotected AES implementation (independently of the used key size) into a white-box protected fixed key implementation according to the scheme of Chow et al. can be achieved in two phases: first, the AES algorithm has to be rewritten and translated as a series of look-up tables and second, secret but invertible encodings have to be applied to all look-up tables in order to build a white-box implementation. The following section will describe this process exemplary for the case of AES-128 as presented in [CEJvO02b], but again subdividing each phase into two steps.

Notation. In the following, we use the bold lower-case letter \( x \) for a single byte of the intermediate round state, \( \hat{k}_r \) for a single byte of a round key, a raising index \( r \) for the current round and lowering indices \( (i,j) \) for the current byte position in the state matrix, where \( i \) denotes the row index and \( j \) the column index. Functions are represented with sans serif fonts. The AES S-box is denoted with \( S \) and the matrix of the \textit{MixColumns} operation is denoted by \( MC \).

Step 1: Partial Evaluation. In the first step, the S-box computation is combined with the preceding addition of the round key. Merging both operations yields into a single look-up table defined as T-box:

\[
T'_{i,j} (x) = S (x \oplus \hat{k}'_{r_{i,j}}) \quad \text{for } 0 \leq i, j \leq 3 \text{ and } 1 \leq r \leq 9
\]

\[
T^{10}_{i,j} (x) = S (x \oplus \hat{k}^{10}_{i,j-i} \oplus \hat{k}^{11}_{i,j}) \quad \text{for } 0 \leq i, j \leq 3
\]

This step results in 160 different key-dependent T-boxes. It should be noted, that the T-boxes of the last round incorporate two bytes of two different round keys. This is due to the missing \textit{MixColumns} operation and the final post-whitening key addition.

Step 2: Matrix Partitioning. A well-known implementation technique for the \textit{MixColumns} operation is to decompose it into four different \( 8 \times 32 \)-bit look-up tables using the matrix partitioning strategy. Eventually, four 32-bit table outputs are added, resulting in the original \textit{MixColumns} transformation. Applying this approach to our previously constructed T-boxes gives us a new set of different TMC tables, where \( MC_i \) denotes the \( i \)-th column of the \( MC \) matrix:

\[
\text{TMC}_{i,j} (x) = MC_i \circ T_{i,j} (x) \quad \text{for } 0 \leq i, j \leq 3 \text{ and } 1 \leq r \leq 9
\]

Finally, this results in 144 different \( 8 \times 32 \)-bit TMC look-up tables and additionally 16 different \( 8 \times 8 \)-bit T-boxes for the last round. Since all look-up tables comprise a small portion of the secret key, they have to be protected against attackers aiming at extracting the secret. For a better illustration, the key-dependent tables can be seen as miniature block ciphers that have to be enhanced by well-known techniques such as diffusion and confusion for protection.
purposes. Before applying randomly chosen invertible nonlinear white-box encodings to the key-dependent tables in order to achieve confusion, diffusion is achieved through the application of linear transformations\(^1\) called mixing bijections.

**Step 3: Mixing Bijections.** To add diffusion in each key-dependent table, two different linear transformations are necessary: an \(8 \times 8\)-bit linear transformation \(L^r_{i,j}\) is inserted before \(\text{TMC}^r_{i,j}\), and a \(32 \times 32\)-bit transformation \(R^r_{i}\) is applied afterwards. In order to cancel out the effect of the transformation \(R^r_{i}\) after the addition of the TMC output values, another untwist table is introduced after each TMC table. This untwist table takes care of canceling the effect of the transformation \(R^r_{i}\) and applying new \(8 \times 8\)-bit transformations \((L^r_{i,j}^{-1})\) to keep the encryption process consistent during all rounds. These transformations can be found by randomly creating linear matrices and checking for invertibility.

**Step 4: Nibble Encodings.** Eventually, nonlinear white-box encodings are applied to all table inputs and outputs. For the sake of efficiency, concatenation of 4-bit nibble encodings were chosen rather than 8-bit byte encodings. Since these nonlinear encodings avoid linear operation over the TMC table outputs, dedicated tables for the XOR operations have to be introduced. These nibble encodings can be found by constructing random 4-bit permutations. All in all, this design strategy results in five different look-up tables that are defined as follows:

\[
\begin{align*}
\mathcal{L}-\text{Ia:} & \quad N_{out} \circ R^1_i \circ \text{TMC}^1_{i,j} \circ (F_{i,j})^{-1} \\
\mathcal{L}-\text{Ib:} & \quad G_{i,j} \circ T^{10}_{i,j} \circ L^{10}_{i,j} \circ (N_{in})^{-1} \\
\mathcal{L}-\text{II:} & \quad N_{out} \circ R^r_{i} \circ \text{TMC}^r_{i,j} \circ L^r_{i,j} \circ (N_{in})^{-1} \\
\mathcal{L}-\text{III:} & \quad N_{out} \circ (L^{r+1}_{i,j})^{-1} \circ (R^r_{i})^{-1} \circ (N_{in})^{-1} \\
\mathcal{L}-\text{IV:} & \quad N_{in} \circ L^2_{i,j} \circ (N_{out})^{-1} \\
\end{align*}
\]

(8 × 32-bit)

(8 × 32-bit)

(8 × 32-bit)

(8 × 32-bit)

(8 × 4-bit)

Combining these tables in their designated way (a single round is depicted in Figure 11.1) results in an encoded fixed-key white-box AES instantiation

\[
\text{AES}'_K = G \circ \text{AES}_K \circ F^{-1},
\]

where \(F^{-1}\) and \(G\) are responsible for external input and output encodings respectively.

**Known Attacks and Vulnerabilities**

Below we briefly outline the known attacks and vulnerabilities of the above presented white-box AES implementation. Some of the threats were already considered during its design. For those, we additionally explain how the attacks were targeted and how the countermeasures were integrated.

\(^1\)Note that originally affine and non-affine transformations were considered. However, since the constant of any affine transformation can be combined with the non-affine mapping, this eventually behaves as linear transformations.
Figure 11.1: White-box implementation of a quarter AES round
11.3 FPGA Implementation

**Code Lifting Attacks.** Since the secret key is hidden and integrated into the white-box implementation, the goal of an attacker is obviously to extract the secret key. However, such fixed-key white-box implementations suffer from another kind of threat where an attacker is not interested in extracting the secret key but instead cloning the entire white-box implementation in order to use it at another place. This threat is known as *Code Lifting* where the entire white-box application is seen as a single key that is cloned and misused by an attacker to encrypt and decrypt data without being in possession of the secret key. To avoid such kind of attacks, external encodings \((F \text{ and } G)\) are introduced, turning an white-box implementation \(E_K\) into an obfuscated encryption function \(E'_K = G \circ E_K \circ F^{-1}\) with hidden external encodings. By pushing the white-box implementation boundaries, the attacker is no longer able to misuse the white-box implementation as long as the external encodings are unknown.

**White-Box Inversion.** Besides cloning the white-box implementation through Code Lifting, inverting the encryption (or decryption) function is another practical issue in particular for white-box implementations of AES. Since the entire algorithm is implemented through look-up tables, any white-box attacker would be able to extract the tables and compute the inverses of all rounds. This allows to turn any implemented encryption (respectively decryption) function into an decryption (respectively encryption) without knowing the secret key. In fact, this issue cannot be prevented but mitigated by external encodings since it prevents the attacker to use the inverted function in a meaningful way. In particular the application of non-invertible external encodings can prevent the inversion of white-box implementations ensuring the property of *one-wayness*.

**Stripping of Nonlinear Encodings.** A first algebraic analysis of the above-explained white-box AES implementation has been presented by Billet et al. [BGE04] which revealed serious vulnerabilities of this design approach by stripping of the nonlinear encodings of the look-up tables and allowing a white-box attacker to efficiently extract the embedded secret key. Later, Michiels et al. [MGH08] generalized this attack for any cipher following the SLT approach. In general, Billet et al.'s approach considers a quarter of the AES round function (depicted in Figure 11.1) as a single \(32 \times 32\)-bit function rather than a decomposition into a series of look-up tables. Following this strategy, the influence of the mixing bijection \(R^t_i\) and any other internal (nonlinear) encoding are canceled out.

It was observed, that with moderate computational effort, the nonlinear encodings at the beginning and end of each quarter AES round can be removed, so that only some (unknown) affine transformation will remain. Applying this technique to three subsequent rounds, thus removing the nonlinear encodings up to an affine part, the secret key eventually can be retrieved with a complexity of at maximum \(2^{30}\) (cf. [BGE04]). Note, however, that this attack is only possible in the setting of white-box adversaries, since an attacker needs to have full access to the tables and control over their inputs and outputs.

11.3 FPGA Implementation

This section briefly outlines the approach of transforming the white-box AES implementation of Chow et al. into an efficient hardware architecture for recent Xilinx Kintex-7 FPGAs. Finally,
we give performance and implementation results on the area and throughput efficiency of the proposed architecture.

### 11.3.1 White-Box Architecture in Hardware

White-Box Cryptography was initially proposed to protect software implementations. In this context we like to remark that bitstream configuration files of FPGA designs are digital binary files that are stored in external memory (that are accessible for an attacker) and thus exposed to very similar threats. Further, the basic idea of white-box implementations is to transform a cryptographic implementation into a series of look-up tables. This perfectly fits the regular structure of FPGAs implementing arrays of look-up tables with programmable interconnections. Hence, we can conclude that FPGAs seem to be a very good fit for cryptographic white-box implementations in hardware.

However, since every individual look-up table of the white-box implementation is different (due to different round keys and randomly chosen encodings), we cannot implement any area-efficient round-based or serialized architecture of the AES algorithm nor reuse any of the look-up tables. Instead, we have to implement an entirely unrolled implementation with every round instantiated separately. Due to the application of BRAM primitives, which have a minimum latency of a single clock cycle, this causes an initial latency of 19 clock cycles (due to 19 stages of 16 parallel look-up tables in the proposed white-box implementation) but in order to increase the throughput it is possible to operate the encryption architecture in a pipeline fashion providing ciphertexts at each clock cycle (after the initial latency).

#### Mapping Tables into CLBs

Besides the implementation of the key depending TMC-Tables and T-boxes, the encoded look-up tables to perform the XOR operations consume a large part of the required storage. Although modern FPGAs provide large amounts of general purpose data storage in terms of BRAM, implementing all look-up tables using these dedicated memory primitives is still not feasible. Therefore, some tables have to be transferred to the general purpose logic in order to fit the design into an FPGA. Since any $8 \times 1$-bit Boolean function can be implemented efficiently into a single slice and each XOR operation and its corresponding look-up table can be decomposed into four different $8 \times 1$-bit functions, it is a natural choice to implement these tables in general purpose logic. In total, each XOR-table can be implemented using four slices equipped with 4 LUTs each, thus in total 16 LUT instances are required (this equals 1024-bit memory). Fortunately, the last round can do without XOR operations, so we only have to implement these tables for 9 rounds. As depicted in Figure 11.1, a quarter round of the AES white-box implementation implements 48 XOR-tables which results in 192 tables per full round and 1728 tables in total.

#### Mapping Tables into Block Memory

The remaining look-up tables can be implemented in BRAM primitives. For most of the tables, except for the T-boxes of the last round, $8 \times 32$-bit functions are implemented which require 8192-bit of memory. Since we can use the BRAM in dual-port mode, two tables can be implemented in a single BRAM which allows us to entirely use the 16-Kbit BRAMs resulting in a very dense
Table 11.1: Area and memory consumption of different table types

<table>
<thead>
<tr>
<th>Look-Up Table/Type</th>
<th>Resources (LUT)</th>
<th>Resources (BRAM)</th>
<th>Memory (Entries)</th>
<th>Memory (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{L}$-Ia</td>
<td>-</td>
<td>8</td>
<td>16</td>
<td>16 384</td>
</tr>
<tr>
<td>$\mathcal{L}$-Ib</td>
<td>-</td>
<td>8</td>
<td>16</td>
<td>4 096</td>
</tr>
<tr>
<td>$\mathcal{L}$-II</td>
<td>-</td>
<td>72</td>
<td>144</td>
<td>147 456</td>
</tr>
<tr>
<td>$\mathcal{L}$-III</td>
<td>-</td>
<td>72</td>
<td>144</td>
<td>147 456</td>
</tr>
<tr>
<td>$\mathcal{L}$-IV</td>
<td>27 648</td>
<td>-</td>
<td>1728</td>
<td>221 184</td>
</tr>
<tr>
<td>Total</td>
<td>27 648</td>
<td>160</td>
<td>2048</td>
<td>536 576</td>
</tr>
<tr>
<td>Utilization (for XC7K160T)</td>
<td>28%</td>
<td>46%</td>
<td>-</td>
<td>40%</td>
</tr>
</tbody>
</table>

and efficient implementation. In total, as depicted in Figure 11.1, 8 different look-up tables with 32-bit output values are implemented in a quarter AES round, thus 32 tables are necessary to build a full round function (except for the last round). In total, 176 different such tables have to be instantiated along with 16 different $8 \times 8$-bit T-boxes for the last round. Note, that all BRAM tables have a similar shape except for the first and last round.

11.3.2 Performance Evaluation

Table 11.1 provides the memory consumption of our white-box implementation of AES-128 broken down to different look-up table types and their implementation size (resources and memory). In total, 536KB of memory are required to implement this white-box implementation on an FPGA, whereby 41% of the memory is required for tables of type $\mathcal{L}$-IV implemented in logic and the remaining 59% of memory is necessary to store tables of type $\mathcal{L}$-I to $\mathcal{L}$-III in BRAMs.

As mentioned before, the design has an initial latency of 19 clock cycles introduced by the BRAM stages. If operated in pipelined mode, this architecture can return one ciphertext per clock cycle after the initial 19 clock cycles. Due to the pipelined architecture and small critical paths, the entire design can operate at a maximum frequency of 100 MHz, resulting in a final throughput of 12.8 Gbit/s. Implementing this on a recent Xilinx Kintex-7 XC7K160T, this design occupies roughly 28% of the available slices and 46% of provided BRAM resources.

11.4 Side-Channel Analysis

In this section, we first summarize recent SCA results on white-box cryptographic implementations in software before we outline the DPA against our hardware implementation. Eventually, we discuss the reasons for our findings and how to avoid such attacks.

11.4.1 Differential Computational Analysis Attack

Recently, Bos et al. introduced a new analysis methodology for cryptographic white-box implementations in [BHMT16] which requires neither knowledge nor possession of the implemented and used look-up tables nor reverse-engineering them during an attack process. The following
section briefly introduces the methodology of DCA attacks in order to extract secret keys from unknown white-box implementations.

**Methodology**

DCA primarily targets software-based white-box implementations. In order to successfully perform a key-recovery attack the following two conditions have to be fulfilled:

1. *The attacker is able to execute and query the white-box implementation several times, with different (randomly chosen) plaintexts.*

2. *Either input or output external encodings are known to the attacker.*

In particular the second requirement is of major importance since it already implies that this attack can be prevented if external encodings are applied and kept secret. However, in practice, at least one encoding (either the initial encoding or the final decoding) usually is known by the user in order to allow a meaningful application of the encryption (or decryption) function. If both aforementioned conditions are fulfilled, assuming that the underlying cryptographic algorithm is known to the attacker, the following three steps can be followed to perform a DCA attack.

**Step 1: Record Multiple Measurements.** It is assumed, that the adversary can execute the white-box implementation in a fully controlled environment. During multiple execution of the encryption algorithm with randomly chosen plaintexts, all accessed memory addresses and any data written to or read from memory are recorded.

**Step 2: Conversion to Ideal Traces.** A certain type of information is extracted from the recorded data. Common examples of promising information are data read from memory (corresponding to the look-up table outputs), data written to stack (intermediate values of the encryption process) or parts of memory addresses (corresponding to inputs of the look-up tables). The extracted data is converted to a format that can be used by common DPA tools. The authors proposed to serialize the recorded data into a binary string and append the results.
11.4 Side-Channel Analysis

according to their temporal occurrence. This final binary string is handled as a kind of side-channel trace that we denote as *Ideal Trace* since it refers to the result of a fully noise-free probing process.

**Step 3: Perform DPA Attack.** Following the concept of classical DPA, by guessing a key byte $k^*$ and knowing the corresponding plaintext bytes $p$, the output bits of the S-box, i.e., $S(p \oplus k^*)$, are predicted. Using these models (8 for each key byte) DPA attacks are performed on the *Ideal Traces* to distinguish the correct key guess among the others.

Although the authors of [BHMT16] reported successful key recoveries, the reason behind such a success has not be clearly stated. Below we first address our observations from an SCA adversary point of view, and later deal with the leakage source.

### 11.4.2 Differential Power Analysis Attack

In this scenario we supposed a gray-box adversary model, where the underlying cryptographic algorithm (e.g., AES) is known, but no information about the type of the implementation and its structure (e.g., white-box or ordinary design) is known to the attacker. Further, we suppose that there is no external encoding in the design, e.g., the gray-box seen by the attacker performs standard AES encryption (or decryption). However, the adversary is able to observe side-channel information (e.g., power consumption) of the implementation while it is operated.

**Measurement Setup.** We made use of a SAKURA-X FPGA board [Lab] equipped with a Kintex-7 XC7K160T FPGA to practically examine the vulnerability of our white-box design with respect to such an SCA adversary. By means of a digital oscilloscope, the side-channel traces have been collected by measuring the voltage drop over a 1 Ω resistor in the $V_{dd}$ path of the FPGA during the operation of the design. The sampling was performed at a rate of 500 MS/s and a bandwidth limit of 20 MHz while the design was running at a stable, jitter-free, but low clock frequency of 3 MHz to mitigate the noise. During the measurement phase, our hardware implementation of white-box AES was provided by fully random plaintexts. A sample power trace, where the rounds (19 clock cycles) are clearly distinguishable, is shown in Figure 11.2.

**Evaluation.** We have collected 10 million power traces of encryptions while the plaintexts were selected randomly. In fact, we have applied several different variants of power analysis attacks including CPA [BCO04], DPA [KJJ99] and collision ones [MME10] with different hypothetical models. The best result has been achieved by means of the classical DPA, which is the same as CPA with single-bit power model. Similar to the case of DCA, for each key byte candidate $k^*$ the output bits of the S-box at the first round, i.e., $S(p \oplus k^*)$, have been predicted and correlated to the power traces. The results of such 8 different CPA attacks on each bit of one of the S-box outputs are shown in Figure 11.3. As shown by the graphics, only one of the attacks (bit 2) is able to recover the secret. We have performed the same attacks on all 16 S-boxes of the first round. Although the attacks on different S-boxes did not show identical results, at least one of the output bits of each S-box led to a successful key recovery, hence full 128-bit key could be recovered.
We would like to note that DCA [BHMT16] is indeed a CPA with single-bit power model, assuming the identity function as the actual leakage model of the device and noise-free measurements. Hence, we have shown that the attack is still feasible in case of imperfect (i.e., noisy) measurements and a more complex side-channel leakage function.

![Figure 11.3: CPA results, S-box output bit model, 10 000 000 traces](image)

11.4.3 Mathematical Foundations

In order to discuss about the reason behind such a leakage, we first need to give the following definitions.

**Definition 11.4.1.** Let \( x = < x_1, ..., x_n > \) and \( \omega = < \omega_1, ..., \omega_n > \) be elements of \( \{0, 1\}^n \). Let \( f(x) \) be a Boolean function of \( n \) variables. Then the Walsh transform of the function \( f(x) \) is a real valued function over \( \{0, 1\}^n \) that can be defined as

\[
W_f(\omega) = \sum_{x \in \{0,1\}^n} (-1)^{f(x) \oplus x \cdot \omega}.
\]
11.4 Side-Channel Analysis

Figure 11.4: Detailed representation of an 8 × 32 look-up table at the first round of our white-box design.

Figure 11.5: Walsh transforms for all 32 functions \( f_{\in \{1, \ldots, 32\}}(\cdot) \) with \( HW(\omega) = 1 \).

**Definition 11.4.2.** If the Walsh transform \( W_f \) of a Boolean function \( f(x_1, \ldots, x_n) \) satisfies \( W_f(\omega) = 0 \), for \( 0 \leq HW(\omega) \leq m \), it is called a balanced \( m \)-th order Correlation Immune (CI) function or an \( m \)-resilient function, where \( HW \) stands for Hamming weight.

For the sake of simplicity, we consider Figure 11.4 as one of the 8-to-32 bit \( L-Ia \) look-up tables used at the first round of our white-box implementation. As stated before, it is supposed that no external encodings exist in the design (or they are known to the adversary), hence we did not draw them in the figure. Let us denote the output of the S-box by \( x \) and the combination of \( MC \) and linear encoding \( R \) and nonlinear 4-to-4 bit encodings by 32 Boolean functions \( f_{\in \{1, \ldots, 32\}}(x) : \{0, 1\}^8 \to \{0, 1\} \). The results of CPA and DCA indicate that at least one of these functions \( f_i(\cdot) \) is not first-order correlation immune. In order to investigate this, we calculated the Walsh transform of all these functions for all \( \omega \in \{0, 1\}^8 \). The results for 8 cases, where \( HW(\omega) = 1 \), are shown in Figure 11.5.

As shown by the graphics, Walsh transform of a couple of functions for two particular \( \omega \) show an extreme imbalance. However, this fact does not guarantee that a CPA or DPA leads to a successful key recovery. To clarify this fact, we suppose that the linear encoding \( R \) and nonlinear 4-to-4 bit encodings are unknown, and for each key candidate \( k^* \) we derive \( f_{\in \{1, \ldots, 32\}}(x) \) by 32-bit output of \( L-Ia(p = S^{-1}(x) \oplus k^*) \). For each key candidate \( k^* \), we again calculated the Walsh transforms for all \( \omega \in \{0, 1\}^8 \). Figure 11.6 represents the results of each \( \omega \): \( HW(\omega) = 1 \) over all key candidates. As shown by the figures, for \( \omega = 2 \) the extreme imbalance of some functions \( f_{\in \{1, \ldots, 32\}}(\cdot) \) for the correct key can be detected amongst that for other key candidates. This indeed justifies why DCA and CPA led to successful key recoveries as this observation perfectly fits to the result of CPA on the same key byte (as shown in Figure 11.3), where similarly only second bit of the S-box output (compatible with \( \omega = 2 \)) led to successful key recovery. It is noteworthy that we have similarly examined all other look-up tables of the first cipher round, and for each of them the Walsh transform of at least one \( \omega; HW(\omega) = 1 \) for the correct key showed extremely high imbalance (compared to that for other key candidates).
Chapter 11 White-Box Cryptography in the Gray Box

Figure 11.6: Walsh transforms for all 32 functions $f_i \in \{1, \ldots, 32\}(\cdot)$ with $HW(\omega) = 1$ for all key candidates $k^* \in \{0,1\}^8$.

should stress that all linear and nonlinear encodings used in our design have been randomly generated as stated in Step 3 and Step 4 of Section 11.2.2.

11.4.4 How to Avoid such Attacks

At the first glance, it can be concluded that if any imbalances is avoided in functions $f_i \in \{1, \ldots, 32\}(\cdot)$, i.e., all $f_i$ to be first-order correlation immune, DPA and DCA can be avoided. However, it should be noted that such a correlation immunity is valid only in case of classical DPA. In other words, if any of the functions $f_i$ has an extremely high imbalance for any $\omega \in \{0, 1\}^8$, that makes it recognizable compared to other key candidates, there exists an attack which can recover the correct key. Such an attack can make use of a power model (or distinguisher) corresponding to that $\omega$. Alternatively, those power analysis attacks which consider the distribution of the leakages, e.g., Mutual Information Analysis (MIA) [GBTP08] which relaxes the power model, can be applied.

In contrary, if many of the functions $f_i$ are $m$-correlation immune (for any arbitrary $m$), this opens another door to recover the key. Suppose that for all key candidates $k^*$ and for all $\omega$ we calculated the Walsh transforms $W_{f_i}$. If we sum up all the imbalances for each key candidate as:

$$\Delta_{k^* \in \{0,1\}^8} = \sum_{\forall \omega \in \{0,1\}^8} \sum_{i=1, \ldots, 32} |W_{f_i}(\omega)| ; k^* = k,$$

the $\Delta_k$ for the correct key candidate might be distinguishable (though minimum). In case of our design (the same look-up table which have been considered above), Figure 11.7 shows $\Delta_k$ for all key candidates, where the correct key is obviously distinguishable. In fact, these results indicate that the linear and nonlinear encodings cannot be arbitrary (randomly) selected. Otherwise, the key can be easily revealed by the above explained procedure. This raises a question as what
should be the characteristics of such random encodings in such a way that these attacks are not applicable. At least, it can be said that \( \forall \omega \) the distribution of Walsh transforms of all \( f_i \) should be not distinguishable from that of other key candidates. But how to define the corresponding characteristics to fulfill such a property is considered as future works.

11.5 Conclusion

In this chapter, we presented the first white-box implementation of AES realized in reconfigurable hardware. Assuming a gray-box adversary model, we have practically examined the resistance of our architecture against side-channel attacks. Unfortunately, we were able to successfully perform attacks using classical DPA. However, our observations approve previous results on software-based white-box implementations and extend these results to hardware implementations and physical side-channel attacks. Finally, we provide a to-date missing thorough mathematical analysis of the underlying reasons that enable attacks on such white-box implementations even assuming a gray-box model in case of a lack of unknown external encodings. Consequently, we identify the static behavior and structure of the white-box implementation as fundamental cause and reason for the observed side-channel vulnerabilities. In practice, a conceivable approach to avoid vulnerabilities of white-box implementations in a gray-box adversary model might be a dynamic update of intermediate encodings. In particular for reconfigurable devices, which offer partial reconfiguration abilities, this might be an interesting approach to make side-channel attacks practically infeasible.
Chapter 12

Randomizing Cryptographic Implementations

In this chapter, based on a joint work with Amir Moradi and Tim Güneysu [SMG17], we investigate the concept of dynamic hardware modifications, i.e., random changes and transformations of cryptographic implementations in order to render higher-order attacks on first-order secure TIs impractical. To this end, we develop a generic methodology which can be applied to (almost) every cryptographic implementation and present a practical instantiation of our methodology that adapts ideas from White-Box Cryptography and applies this construction to a first-order secure TI of the PRESENT cipher. Eventually, we show that dynamically updating cryptographic implementations during operation provides the ability to avoid higher-order leakages to be practically exploitable.

Contents of this Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1</td>
<td>Introduction</td>
<td>143</td>
</tr>
<tr>
<td>12.2</td>
<td>Background</td>
<td>145</td>
</tr>
<tr>
<td>12.3</td>
<td>Methodology</td>
<td>145</td>
</tr>
<tr>
<td>12.4</td>
<td>Case Study: PRESENT Threshold Implementation</td>
<td>148</td>
</tr>
<tr>
<td>12.5</td>
<td>Practical Side-Channel Evaluation</td>
<td>152</td>
</tr>
<tr>
<td>12.6</td>
<td>Conclusion</td>
<td>155</td>
</tr>
</tbody>
</table>

12.1 Introduction

SCA uses information leakage by measuring physical device internals, e.g., timing [Koc96], power consumption [KJJ99] or electromagnetic emanations [AARR02], to extract cryptographic secrets. Modern side-channel countermeasures are classified either as hiding or masking [MOP07]. While hiding countermeasures aim to decrease the SNR in order to hide information leakage in random noise, masking countermeasures tackle information leakage using secret sharing and multi-party computation techniques. The idea of TI has been developed based on Boolean masking in particular to target hardware implementations [NRR06]. However, the initial concept of TI was only suitable to counteract first-order side-channel leakages, still allowing attacks using higher-order statistical moments to successfully recover cryptographic secrets. Naturally, HO-TI have been proposed to solve this problem [BGN+14h]. Despite, HO-TI might be limited to univariate scenarios [RBN+15] as well as they come with increased time overhead and area demands due to the ever increasing number of minimum shares for higher-order protection.
Therefore, combining first-order secure TI with hiding countermeasures to achieve (practical) higher-order protection might be an alternative solution. More precisely, although our findings in the previous chapter show that current White-Box Cryptography is vulnerable to SCA and suffers from its static behavior and structure, we can adopt the basic concept of encoded functions and enhance it by dynamic randomization in order to create a generic hiding countermeasure based on dynamic hardware modifications.

12.1.1 Contribution

Our contribution in this chapter is twofold: First, we present a generic approach to change the representations of cryptographic implementations dynamically in order to introduce non-static behavior. Our methodology can be applied to (almost) every cryptographic implementation and circuit independent of the cryptographic algorithm or scheme. Our approach uses random substitution of basic elements along with random encoding of intermediate connections and offers high flexibility and scalability of attack complexities depending on the used level of abstraction and granularity of the underlying circuit. Second, we investigate and analyze a specific instantiation of our approach to randomize a TI. In particular, we are going to examine a first-order PRESENT TI as a case study which is implemented in reconfigurable hardware. The randomization of intermediate signals, in terms of random nonlinear 4-bit encodings, is chosen dynamically during run time and injected into each implemented look-up table in order to substitute them by different representations. In particular, this approach adapts ideas and techniques from the area of White-Box Cryptography, although we want to emphasize that we do not aim to achieve resistance against attacks in the white-box adversary model. Eventually, we conduct practical side-channel measurements for our case study. Using a leakage assessment methodology, we focus on effects of our countermeasure on higher-order statistical properties and moments and show that our approach can increase the protection against higher-order side-channel attacks from a practical point of view.

12.1.2 Previous Work

Although the threat of side-channel attacks is well known, many cryptographic devices are vulnerable to side-channel analysis due to their static design and behavior which allows attacks based on statistical and differential analysis. Introducing dynamic behavior in terms of ever-changing and morphing implementations and circuits could help to overcome these problems. However, this is not a trivial task and poses big challenges to designers of cryptographic implementations in particular using static hardware devices. In recent years, several research into this direction has been performed and published but still existing solutions are at an early stage and have to face many difficulties. In 2008, Mentens et al. [MGV08] introduced a first work for using dynamic reconfiguration of modern FPGAs as countermeasure against power and fault attacks. However, their solution had to struggle with slow reconfiguration times as well as too small complexity which still allowed efficient analysis and attacks. Besides, their solution specifically targeted reconfigurable hardware which provides dynamic reconfiguration features. Moradi and Mischke [MM13] examined the opportunities of using dual ciphers as alternative representations (in particular for AES) in order to achieve protection against side-channel attacks. Though, dual ciphers maintain structural properties of the original representation which again could be exploited using statistical analysis.
In Chapter 10, we proposed the application of affine equivalence representations of cryptographic S-boxes to change the cipher implementation dynamically during run time, based on our results published in [SMG15a]. Although the complexity of this approach is quite high, it exploits very specific properties of the cryptographic components, i.e., the nonlinear S-box, so that this approach cannot be generically applied to cryptographic implementations.

12.2 Background

Before we present our generic methodology of dynamic hardware modifications, this section briefly reviews the theory of directed graphs and their most important properties with regard to our concept.

12.2.1 Directed Graphs

Directed Graphs (or digraphs) are used for many applications in order to abstractly model a certain problem and find according solutions. In general, a graph is a set of nodes that are connected by some edges. For a directed graph, the edges are provided with a certain direction.

Definition 12.2.1. A directed graph or digraph is an ordered pair of sets $G = (V, A)$ where $V$ is a set of vertices and $A$ is a set of ordered pairs $a_{ij} = \langle v_i, v_j \rangle$ (called arrows or directed edges) with $v_i, v_j \in V$.

In particular, each vertex has a certain number of connected edges. Due to the direction of the edges, we can distinguish between edges that arrive at a vertex and edges that leave a vertex. The number of arriving edges is given by the in-degree of a node, whereas the number of leaving edges is given by the out-degree.

Definition 12.2.2. In a directed graph, the in-degree $\text{deg}^+(v)$ and the out-degree $\text{deg}^-(v)$ of a vertex $v \in V$ count the number of directed edges connecting to and from a vertex respectively. It holds, that $\sum_{v \in V} \text{deg}^+(v) = \sum_{v \in V} \text{deg}^-(v) = |A|$.

Eventually, every node (connected to a digraph) has to have at least one arriving or leaving edge. In case the node has an in-degree of zero, it is called source, since it only serves as starting point for several edges. Similarly, a node without any leaving edges is called sink, since it is only an ending point for some edges. In the following, we consider the source nodes as starting points of our directed graph, whereas the sink will be the final points.

12.3 Methodology

In this section, we introduce our methodology to dynamically update and randomize cryptographic implementations using a generic approach. We first state some important observations that directly lead to a generic representation of the problem. This is followed by an algorithmic solution to achieve dynamic updates of cryptographic implementations.
12.3.1 Generic Approach

In general, our generic approach can be applied to any cryptographic implementation. However, the provided physical platform has to allow some changes of the implementation during run time. Since we want to focus on hardware implementations throughout this chapter, we particularly target reconfigurable hardware in terms of FPGA. Eventually, we present a solution that achieves on-the-fly dynamic randomization of cryptographic implementations.

Observation 12.1. Any cryptographic implementation can be represented as network or sequence of modular or atomic functions subsequently applied on an internal state.

Consequently, we can model any cryptographic implementation as a directed graph. Depending on the level of abstraction and the desired granularity (e.g., system level, gate level, etc.), each node of the graph represents a single or multiple modular and atomic functions of the algorithm. Besides, the edges which connecting the nodes in a certain direction represent the data flow of the internal state.

Observation 12.2. Any cryptographic implementation can be modeled by different but equivalent directed graphs.

In general, the numbers of nodes and edges required to model a cryptographic implementation is not determined and particularly not limited by an upper bound. Principally, we can add new nodes and edges arbitrarily to the graph to find new representations (with sufficient complexity). However, we still have to maintain and ensure correctness of the overall implementation.

12.3.2 Morphing Algorithm for Cryptographic Implementations

Based on this observations, we developed a generic algorithm to morph a digraph of a cryptographic implementation into an equivalent but encoded digraph while still maintaining correctness of the implementation.

According to Algorithm 3, each arrow \( \langle v_i, v_j \rangle \) of a digraph is replaced by an encoded directed edge. For this purpose, both adjacent vertices have to be replaced as well. The starting vertex \( v_i \) is replace such that it not only performs its originally provided function but in addition performs an encoding function \( f \) to the state. In order to maintain correctness of the implementation, the ending vertex \( v_j \) has to cancel the applied encoding using the inverse (decoding) function \( f^{-1} \) before performing its original function to the state.

12.3.3 Applicable encoding functions

In this section, we will briefly discuss properties and requirements on encoding functions that are applicable within our algorithm. First of all, the encoding function should be a randomly drawn function in order to perform a randomization of the implementation during the update. However, each encoding function has to fulfill a few minimal requirements and has to provide some properties to be compatible with our methodology. Obviously, the encoding function has to be injective, i.e., it has to be information preserving in order to allow a correct operation of the original implementation. Apart from that, input and output sizes of the encoding functions will depend on the desired granularity of the algorithm and can differ as long as the output size
12.3 Methodology

Algorithm 3: Morphing algorithm for cryptographic implementations

**Input**: $G = (V, A)$: digraph representing a cryptographic implementation.

**Output**: $G^* = (V^*, A^*)$: digraph representing an encoded cryptographic implementation.

$G^* = (V^*, A^*)$: $V^* \leftarrow V$, $A^* \leftarrow A$

for $\forall v_i \in V^*$ do
  $D \leftarrow \emptyset$
  $s \leftarrow f(v_i)$, $V^* \leftarrow V^* \setminus \{v_i\}$
  for $\forall v_j \in V^*$ do
    if $a_{ij} \in A^*$ then
      $D \leftarrow D \cup f^{-1}(v_j)$
      $V^* \leftarrow V^* \setminus \{v_j\}$, $A^* \leftarrow A^* \setminus \{a_{ij}\}$
    end
  end
  for $\forall d_i \in D$ do
    $V^* \leftarrow V^* \cup \{s, d_i\}$, $A^* \leftarrow A^* \cup \{s, d_i\}$
  end
end
return $G^*$

is at least the input size. Besides, the chosen encoding function can have any complexity (but still should be reasonable efficient). Possible realizations of encoding functions could be: linear functions [XL09], nonlinear bijections (like S-boxes) [CEJvO02b], or any other instance which meets the requirements.

12.3.4 Verification and semantic equivalence checking

Since our methodology should not affect the correctness of the final result of the original implementation, we have to ensure semantic equivalence of the randomized implementations. Therefore, our approach has to include checking and verification steps. As mentioned before, the randomly drawn encoding functions have to meet minimal requirements which has to be checked and verified continuously during the operation. Correctness of the final result, i.e., semantic equivalence of the randomized implementation, is ensured by only encoding single edges (or small paths\(^1\)) and including the inverse decoding function at the same time.

\(^1\)Given for instance a linear operation within a cryptographic implementation (e.g., MixColumns of the AES algorithm) and the application of linear encoding functions would allow to keep encoded intermediate values. However, the decoding function then has to consider the inversion of the linear operation as well.
12.4 Case Study: PRESENT Threshold Implementation

Throughout this section, we present a practical realization of our proposed countermeasure using an encoded PRESENT TI as case study. Before investigating the feasibility of our approach in terms of hiding higher-order side-channel leakage, we give a detailed description of our practical architecture realized on a modern Xilinx FPGA and elaborate our design strategy.

12.4.1 Adversary Model

Although our practical instantiation employs certain ideas and concepts of White-Box Cryptography in terms of using encoded look-up tables to hide secret key material, we want to emphasize that we still do not consider adversaries of the white-box model. It is obvious that every adversary who has full access and control of the execution environment can circumvent our proposed countermeasures in order to extract secret keys from the implementation using more powerful attacks, e.g., an algebraic analysis of the look-up tables. However, we therefore only consider adversaries of the gray-box model, i.e., adversaries that still can access the implementation but can only gain helpful information through side-channel leakage.

12.4.2 Design Considerations

PRESENT [BKL+07] is a lightweight symmetric block cipher based on a block size of 64 bits. In particular, it is a SPN with 31 rounds. It provides two different key sizes (80 bit or 128 bit) and derives 32 different 64-bit round-keys based on the initial key. Since nowadays, it is advised against using 80-bit keys, we opted to implement and focus on PRESENT-128.

Threshold Implementation of PRESENT:

Our implementation is based on the first TI that was presented in [PMK+11]. In particular, we apply the decomposition of the S-box into two quadratic functions \( g \) and \( f \) that was proposed by Poschmann et al. in order to benefit from the minimal number of shares (i.e., \( m = n = 3 \)). Since the permutation of the PRESENT cipher is a linear function, it can be applied to each share individually and without modification. Due to the decomposition of the S-box, additional register stages have to be placed in between \( g \) and \( f \) in order to prevent side-channel leakage caused by glitches. The final structure of the first-order TI of the PRESENT S-box is shown in Figure 12.1.
12.4 Case Study: PRESENT Threshold Implementation

Before instantiating and implementing our proposed algorithm taking the example of a first-order secure PRESENT TI, we have to find an architecture which supports dynamic updates of sub-functions or components and can be implemented on an FPGA. Given the basic structure of a TI of the PRESENT S-box as shown in Figure 12.1 we chose the component functions as basic building blocks that have to be updated on-the-fly. Besides, we opted to implement each function as look-up tables because it is a natural choice for FPGAs but also allows fast updates.

Starting from this, the PRESENT TI can be implemented as network of look-up tables, each operating on 4-bit nibbles of the internal state. In a next step, the output of each look-up table is encoded using a nonlinear 4-bit bijection. In order to maintain correctness, all subsequent look-up tables have to apply the according decoding function before being evaluated, i.e., the original table has to be combined with the according inverse bijection. In general, this approach reflects basic ideas and concepts of White-Box Cryptography as initially proposed by Chow et al. in [CEJvO02a, CEJvO02b].

However, this strategy has some important implications that effect the final hardware architecture. First, the secret key has to be known during design time since it is included within the look-up tables. Hence, the (shared) key is fixed and combined with the look-up tables of the first layer of the TI S-box. Second, since the permutation layer is a linear functions which operates on single bits, we cannot perform the permutation on 4-bit encoded values. Instead, we have to implement the permutation layer as sequence of look-up tables that decode and re-encode the nibbles while performing the original permutation.

Eventually, our encoded TI is implemented using different look-up tables for each sub-function and all rounds. However, this complicates the task of implementing our design efficiently using an round-based approach. None the less, modern FPGAs provide useful features that allow an efficient implementation (as presented in Section 12.4.3).
Table 12.1: Area consumption of our hardware architecture

<table>
<thead>
<tr>
<th>Module/Component</th>
<th>Resource Utilization</th>
<th>Area (Slices)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Logic (LUT)</td>
<td>Memory (FF) (LUTRAM) (BRAM)</td>
</tr>
<tr>
<td>Control Logic</td>
<td>11</td>
<td>24</td>
</tr>
<tr>
<td>Round Function</td>
<td>96</td>
<td>0</td>
</tr>
<tr>
<td>(g)-Layer</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(f)-Layer</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(p_1)-Layer</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(p_2)-Layer</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Reconfiguration</td>
<td>3129</td>
<td>3222</td>
</tr>
<tr>
<td>Context Engine</td>
<td>22</td>
<td>44</td>
</tr>
<tr>
<td>Encoding Engine</td>
<td>2880</td>
<td>2880</td>
</tr>
<tr>
<td>Randomness Generator</td>
<td>136</td>
<td>256</td>
</tr>
<tr>
<td>Total</td>
<td>3236</td>
<td>3246</td>
</tr>
</tbody>
</table>

Dynamic Update of Encodings

So far, our TI is encoded statically using arbitrary nonlinear functions applied during design time. However, in order to perform dynamic randomization during operation time, we want to modify these initial encodings. Therefore, in general, we have to find solutions for the following two issues:

1. How to find or compute random nonlinear functions on-the-fly?
2. How to inject random nonlinear functions into our hardware implementation during run time?

Random 4-bit nonlinear functions, i.e., a random permutations of the sequence \(\{0, 1, \ldots, 15\}\) can be generated using a linear-time algorithm using swapping operations and sampling uniform random numbers [VMKS12]. Although the permutation generation is slightly biased, this effect can be neglected in the context of side-channel analysis.

Since our encoded TI is implemented as network of look-up tables, injecting random nonlinear functions can be realized as table re-computation and re-ordering. In particular, we can apply arbitrary functions to the output of a table by replacing each table entry by the according encoded value. The decoding function can be applied to the input of a table by re-ordering the table entries according to the decoded address value. Fortunately, this procedure is independent of the previous injection of random functions, i.e., if we first apply a random function \(n_1\) follow by a second function \(n_2\) this is the same as applying another function \(n_3\) with \(n_3 = n_2 \circ n_1\). Hence, we can continually update our implementation using random nonlinear functions without increasing the size of our implementation by just performing table re-computations and re-orderings.

Eventually, for the given PRESENT implementation, we have to update 5904 4-bit encodings per encryption in order to perform a full dynamic hardware modification process. Since there are 16! different 4-bit encodings, the final randomization complexity of our methodology (for the given case study) is about \(2^{35}\).
12.4 Case Study: PRESENT Threshold Implementation

Table 12.2: Comparison of different PRESENT Hardware Architectures

<table>
<thead>
<tr>
<th>Scheme/Implementation</th>
<th>Logic (LUT)</th>
<th>Memory (LUTRAM)</th>
<th>Latency (cycles)</th>
<th>Freq. (MHz)</th>
<th>Throughput (MBit/s)</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st-order TI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KATAN</td>
<td>34</td>
<td>96</td>
<td>-</td>
<td>-</td>
<td>226</td>
<td>26.42</td>
</tr>
<tr>
<td>PRESENT</td>
<td>808</td>
<td>384</td>
<td>-</td>
<td>-</td>
<td>64</td>
<td>413.22</td>
</tr>
<tr>
<td>2nd-order TI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KATAN</td>
<td>65</td>
<td>180</td>
<td>-</td>
<td>-</td>
<td>322</td>
<td>37.69</td>
</tr>
<tr>
<td>PRESENT</td>
<td>2245</td>
<td>1680</td>
<td>-</td>
<td>-</td>
<td>128</td>
<td>406.92</td>
</tr>
<tr>
<td>Affine Equivalence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRESENT</td>
<td>1834</td>
<td>742</td>
<td>1</td>
<td>64</td>
<td>112</td>
<td>224.72</td>
</tr>
<tr>
<td>Glitch-Free Duplication</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KATAN</td>
<td>114</td>
<td>548</td>
<td>-</td>
<td>-</td>
<td>439</td>
<td>25.68</td>
</tr>
<tr>
<td>PRESENT</td>
<td>5442</td>
<td>12672</td>
<td>-</td>
<td>704</td>
<td>459</td>
<td>458.09</td>
</tr>
<tr>
<td>Dynamic Hardware Mod.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRESENT</td>
<td>3236</td>
<td>3246</td>
<td>1952</td>
<td>192</td>
<td>125</td>
<td>153</td>
</tr>
</tbody>
</table>

12.4.3 Practical Implementation on Reconfigurable Hardware

The deliberate application of modern reconfigurable hardware in terms of a Xilinx Kintex-7 FPGA provides several interesting advantages and allows a practical evaluation and implementation in order to confirm the feasibility of the proposed approach. In particular, the selected Kintex-7 XC7K160T FPGA implements roughly 12 Mb of BRAM in the form of 325 individual memory instances, each providing 32-Kb of general purpose memory as well as a true dual-port feature. Note that the dual-port option is of particular importance for the dynamic update of our implementation since we can use one port solely to perform the cryptographic operations whereas the second port is used to perform the dynamic table re-ordering and re-computation.

Besides, since all look-up tables of our architecture are $256 \times 4$-bit tables, each BRAM primitive could store up to 32 different look-up tables. Fortunately, PRESENT has only 31 different rounds, so we can arrange tables of the same operation but different rounds in the same BRAM instance. This strategy yields a round-based hardware architecture as presented in Figure 12.2. Moreover, since each BRAM still provides enough memory to store another table we can use this free table entry to store an updated table. Hence, after performing the table re-ordering and re-computation and storing the updated table in the free segment, a context switch is performed, i.e., the storage of the old table is released and the updated table is applied during operation. But since the update is performed through the second port while the first port is continuously used for operation, our strategy does not affect the overall performance.

Table 12.1 provides the implementation numbers of our design, including control logic and a reconfiguration unit that generates new random 4-bit encodings on-the-fly. Obviously, a lion’s share of the used resources is necessary to implement the encoding generation. Basically, the round function can be implemented in 192 BRAM instances – the remaining logic in terms of LUTs is necessary to control and operate the table update using the second port of the BRAM. Eventually, the control logic implements a small FSM that controls both the round function and the reconfiguration engine and provides an interface for external access and control.
Chapter 12 Randomizing Cryptographic Implementations

12.4.4 Comparison

In Table 12.2, we provide a comparison of different approaches to achieve higher-order side-channel resistance (except for the 1st-order TI) by the example of PRESENT. Obviously, our approach offers competitive results, both in terms of performance and area utilization, although it has an increased demand for BRAM instances. Still, the security of our proposed countermeasure may not only be limited to second-order attacks but it may also affect higher-order leakages, hence providing better security than a 2n-order TI (at least from a practical point of view).

12.5 Practical Side-Channel Evaluation

We evaluated side-channel information of our design implemented on a physical device using a SAKURA-X FPGA platform [Lab] which provides a Xilinx Kintex-7 XC7K160T FPGA for practical side-channel evaluations using the power consumption of the device. Measuring the voltage drop over a 1Ω resistor in the Vdd path of the FPGA using a digital oscilloscope with a sampling rate of 625 MS/s, 250 MHz bandwidth limitation, and a stable, jitter-free clock frequency of 24 MHz, we could practically examine vulnerabilities of our proposed design.
12.5 Practical Side-Channel Evaluation

12.5.1 Results

In this section we provide practical evaluation results using the non-specific \( t \)-test on the first, second and third statistical order. Besides, we include the evolution of the absolute maximum of the \( t \)-test over the number of used traces. In total, we performed measurements and evaluations for three different evaluation profiles: first, reference measurements without sharing (i.e., all-zero masks) and omitted dynamic update, and second, measurements using shared plaintexts but still omitting dynamic update of the implementation and finally, measurements using shared values and including our proposed countermeasure in terms of dynamically updating and randomizing the implementation.

Profile 1:

Before evaluating the feasibility and effectiveness of our proposed approach, we have to ensure the correctness of our implemented first-order TI using reference measurements. In order to provide such a reference, we measured one million power traces while the PRNG that generates the random masks for sharing and random encodings was disabled, i.e., all masks were set to zero and the dynamic update was omitted. We expect to detect and observer leakage on all considered statistical orders which is confirmed by our evaluation results shown in Figure 12.3. One the left-hand side, we provide the results of the non-specific \( t \)-test for the first, second and third order after measuring and evaluating the total number of 100,000 traces while on
Chapter 12 Randomizing Cryptographic Implementations

the right-hand side, the development of the absolute maximum for the t-test on each statistical order over the number of evaluated traces is shown.

Profile 2:
Starting from the reference evaluation, we perform measurements for a second profile where the PRNG is active in order to generate random masks (i.e., the sharing is active) but still the dynamic update is omitted. Since we have implemented a first-order TI, we expect to detect leakage on all statistical orders but the first. Obviously, our TI is implemented correctly and behaves as expected which is confirmed by the evaluation results shown in Figure 12.4 (although we only detect second-order leakage). Again, the left-hand side shows the t-test results (after measuring and evaluating 100 million traces) while the right-hand side shows the evolution of the absolute maximum of the t value over the number of traces.

Profile 3:
Eventually, we extend the previous measurement profile by applying our proposed approach in order to hide higher-order side-channel leakages by continuously performing dynamic updates of the look-up tables of our implementation. Again, we do not expect to detect any first-order leakage due to the application of a first-order TI but moreover the leakage detectable at higher statistical orders should be prevented as well. The evaluation results shown in Figure 12.5 confirm the correctness of these assumptions since we could not detect any leakage after
measuring 1.5 billion power traces – neither at the first, second nor third statistical order – which hence also confirms the effectiveness of our proposed approach.

12.6 Conclusion

In this chapter, we have presented a generic strategy and methodology in order to apply dynamic and random updates to cryptographic implementations and circuits in order to hide higher-order side-channel leakages. Using a case study based on a first-order PRESENT TI and a random updates based on nonlinear encodings, we have shown the feasibility and practicability of proposed concept using side-channel power measurements and applying the state-of-the-art leakage assessment methodologies. Eventually, we can conclude that our methodology presents a viable alternative to building HO-TIs and convinces by its generality and scalability.
Part IV

Conclusion
Chapter 13

Conclusion and Future Work

In this chapter we provide a brief summary of the results and a conclusion for this thesis. In addition, we provide new impulses for further research and future work in the area of efficient implementations on reconfigurable devices and hardware agility for physical protection. In particular, since this thesis mainly focuses on protection against passive side-channel attacks, further research on protection against active Fault Injection Attacks (FIAs) or Reverse Engineering (RE) appears to be necessary.

Contents of this Chapter

13.1 Conclusion .................................................. 159
13.2 Directions for Future Work .............................. 160

13.1 Conclusion

In this thesis, we have presented novel results on efficient architectures tailored for reconfigurable devices implementing both symmetric and asymmetric cryptographic primitives. Moreover, in the second part of this thesis, we have advanced the seminal results in the field of cryptographic hardware agility for physical protection and presented novel approaches using structural and algorithmic reconfiguration on re-programmable hardware devices.

Throughout the first part of this thesis, we took a closer look at efficient architectures for AES as the predominant symmetric cipher and Curve25519 and Curve448 as ECC candidates for the next generation of TLS. By exploring specific properties of these algorithms and exploiting useful features and resources of modern FPGA, we could cover a wide range of optimization goals from low-area to high-performance. To this end, our results emphasize that careful engineering and tailoring of hardware architectures for the targeted platform enable novel applications for cryptographic primitives that have not been intended during their specification and design process opening up completely new opportunities and applications.

The second part of this thesis is dedicated to the notion of cryptographic hardware agility for physical protection. In particular, we have pursued two different approaches in order to achieve and realize hardware-agile cryptographic implementations. In a first attempt, we exploited specific properties and provided reconfiguration features of modern FPGAs on a structural level though our results clearly demonstrate limitations and disadvantages of this approach. By abstraction of the problem to the algorithmic level, we could present novel and promising approaches with application for particular cryptographic algorithms as well as generic methods.
that are applicable for virtually any cryptographic implementation. All in all, based on thorough practical evaluations, we have shown that hardware agility in combination with existing masking countermeasure can improve the physical resistance of hardware architectures against SCA making side-channel attacks inefficient or unfeasible from a practical point of view.

13.2 Directions for Future Work

In this section, we discuss general ideas and possible opportunities to continue research and extend our contributions in the different fields of research that we covered in this thesis.

13.2.1 Efficient Implementations

Newly proposed algorithms, changing parameters and conditions, novel applications or new hardware platforms continually require the design of novel, efficient and tailored architectures and implementations. In the following, we briefly summarize potential challenges for the near future with respect to efficient hardware implementations.

New Algorithms and Schemes. In recent years, Lightweight Cryptography has gained a lot of attention and many novel algorithms have been proposed in this area [BKL+07, GPPR11, SIH+11, BCG+12, BSS+13, BJK+16a]. Besides cryptographic strength and resistance against cryptanalytic attacks, these algorithms have to provide lightweight properties and have to be implemented and evaluated on various (hardware and software) platforms with respect to area-efficiency, performance and resistance against physical attacks. In addition, cryptographic competitions, such as the selection of the Advanced Encryption Standard in 2001, or the ongoing Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) constantly provide engineers and researches with new algorithms and cryptographic schemes that have to be implemented on different platforms for various applications and optimization goals.

New Architectures and Devices. On the other hand, hardware platforms and physical devices, such as µCs, FPGAs or ASICs, are constantly improved and modified to provide better performance, higher efficiency and additional capabilities or to reduce costs. In that sense, engineers and researchers constantly have to explore these new opportunities and features for their purposes and applications. Often, new features and capabilities are added and intended for different applications, but prove to be efficient and beneficial for unconsidered problems.

13.2.2 Hardware Agility and Physical Protection

Throughout this thesis, we have proven that hardware agility for cryptographic implementation in combination with common countermeasures, such as masking and hiding, provides improved resistance against physical threats. However, we have limited our research to passive attacks based on SCA but did not investigate other threats such as FIA or RE of hardware designs and Intellectual Properties (IPs). In particular, cryptographic hardware agility may proof as an appropriate solution to prevent not only SCA but FIA and RE at the same time while
reducing costs and improving efficiency in comparison to composed solutions based on individual countermeasures for each threat.

**Protection Against Fault Injection Attacks.** Besides passive observation of side-channel information, active injection of faults to cryptographic primitives during operation pose a serious threat [BS97]. In particular, Laser Fault Injection (LFI) [SA02] has established as a viable field of research. Due to tight control over time and space of induced faults, LFI provides a powerful tool for physical attacks and potential adversaries. Consequently, constant restructuring and reorganization of cryptographic implementations can help to render FIA impossible since then the required precision of the fault injection will be beyond the control of the adversary. At the same time, passive side-channel attacks will be prevented simultaneously without the risk of undesired interdependencies between different kinds of countermeasures.

**Obfuscation and Protection Against Reverse Engineering.** In recent years, IP protection and prevention of RE has become increasingly important and has gained a lot of scientific attention. In particular in the presence of hardware Trojans and modifications or tampering of physical hardware, protection of hardware IP cores that are implemented in untrusted environments and on various platforms is becoming increasingly challenging. Similar to code morphing obfuscation techniques for protection against static or dynamic analysis and reverse-engineering, hardware agility can incorporate different obfuscation techniques, e.g., Garbled Circuits [Yao86, BHR12] or White-Box Cryptography [CEJvO02a, CEJvO02b], to prevent analysis and reverse-engineering of sensitive hardware circuits.

**Hardware Virtualization Techniques.** Modern security concepts and solutions often rely on virtualization techniques to prevent unauthorized access to sensitive information. Consequently, hardware virtualization techniques, such as virtual FPGAs [FHG+11, SSH13, KSW+17] can be deployed in order to e.g., prevent evaluation and accurate modeling of physical hardware (to protect against template-based side-channel attacks) or to allow fast and efficient reconfiguration or customization of the underlying (virtual) hardware. Hence, efficient hardware virtualization opens up entirely new possibilities and opportunities for the concept of hardware agility in order to protect against physical attacks.
Part V

Appendix
Chapter 14

Component functions of the PRESENT S-box

14.1 Uniform TI of $Q_{12}$ using 3 shares

\[ y^1 = f_{Q_{12}}^{2, 3} (x^2, x^3) = f_{Q_{12}}^{2, 3} ((x_0, x_1, x_2, x_3, x_0, x_1, x_2, x_3)) = \langle y_0^1, y_1^1, y_2^1, y_3^1 \rangle \]
\[ y_0^1 = x_0^2, \]
\[ y_1^1 = x_1 + x_1 x_3^3 + x_2 x_3^2 + x_3 x_1^2 + x_3 x_2^2 + x_2 x_3^2, \]
\[ y_2^1 = x_2 + x_1 x_3 + x_2 x_3^3 + x_3^2, \]
\[ y_3^1 = x_3^2 \] (14.1)

\[ y^2 = f_{Q_{12}}^{3, 1} (x^3, x^1) = f_{Q_{12}}^{3, 1} ((x_0, x_1, x_2, x_3, x_0, x_1, x_2, x_3)) = \langle y_0^2, y_1^2, y_2^2, y_3^2 \rangle \]
\[ y_0^2 = x_0^3, \]
\[ y_1^2 = x_1^3 + x_1 x_3^3 + x_2 x_3^2 + x_3 x_1^2 + x_3 x_2^2 + x_2 x_3^2, \]
\[ y_2^2 = x_2^3 + x_1 x_3 + x_2 x_3^3 + x_3^2, \]
\[ y_3^2 = x_3^3 \] (14.2)

\[ y^3 = f_{Q_{12}}^{1, 2} (x^1, x^2) = f_{Q_{12}}^{1, 2} ((x_0, x_1, x_2, x_3, x_0, x_1, x_2, x_3)) = \langle y_0^3, y_1^3, y_2^3, y_3^3 \rangle \]
\[ y_0^3 = x_1^1, \]
\[ y_1^3 = x_1^2 + x_2 x_3^3 + x_3 x_1^2 + x_3 x_2^2 + x_2 x_3^2, \]
\[ y_2^3 = x_1^2 + x_1 x_3 + x_2 x_3^3 + x_3^2, \]
\[ y_3^3 = x_3^1 \] (14.3)

14.2 Uniform TI of $Q_{294}$ using 3 shares

\[ y^1 = f_{Q_{294}}^{2, 3} (x^2, x^3) = f_{Q_{294}}^{2, 3} ((x_0, x_1, x_2, x_3, x_0, x_1, x_2, x_3)) = \langle y_0^1, y_1^1, y_2^1, y_3^1 \rangle \]
\[ y_0^1 = x_0^2 + x_1^2 x_2^3 + x_2^3 x_3^1 + x_1^2 x_3^3, \]
\[ y_1^1 = x_1^2 + x_2^2 x_3^3 + x_3 x_2^3 + x_2 x_3^3, \]
\[ y_2^1 = x_2^2, \]
\[ y_3^1 = x_3^3 \] (14.4)
Chapter 14 Component functions of the PRESENT S-box

\[ y^2 = r_{Q_{299}}^{1,1}(x^3, x^1) = r_{Q_{299}}^{3,1}(x^3, x^1, x^2) = (y_0^2, y_1^2, y_2^2, y_3^2) \]
\[ y_0^2 = x_0^3 + x_1^3x_2^3 + x_3^3x_1 + x_1^2x_3 \]
\[ y_1^2 = x_1^3 + x_2^3x_3^3 + x_3^3x_2 + x_2x_3^3 \]
\[ y_2^2 = x_2^3 \]
\[ y_3^2 = x_3^3 \] (14.5)

\[ y^3 = r_{Q_{299}}^{1,2}(x^1, x^2) = r_{Q_{299}}^{3,2}(x^1, x^1, x^2, x^3) = (y_0^3, y_1^3, y_2^3, y_3^3) \]
\[ y_0^3 = x_0^1 + x_1^1x_3^1 + x_3^1x_1 + x_1x_3^1 \]
\[ y_1^3 = x_1^1 + x_2^1x_3^1 + x_3^1x_2 + x_2x_3^1 \]
\[ y_2^3 = x_2^1 \]
\[ y_3^3 = x_3^1 \] (14.6)

14.3 Uniform TI of \( Q_{299} \) using 3 shares

\[ y^1 = r_{Q_{299}}^{2,3}(x^2, x^3) = r_{Q_{299}}^{3,3}(x^2, x^2, x^3) = (x_0^3, x_1^3, x_2^3) = (y_0^1, y_1^1, y_2^1, y_3^1) \]
\[ y_0^1 = x_0^2 + x_1^2x_3^2 + x_3^2x_0 + x_0^3x_3 + x_2^3x_1 + x_1^2x_2^3 + x_2^3x_2^3 \]
\[ y_1^1 = x_1^2 + x_0^2x_2^2 + x_2^2x_3^2 + x_3^2x_0 + x_0^3x_3 + x_1^3x_2^3 + x_2^3x_2^3 \]
\[ y_2^1 = x_2^2 + x_1^2x_3^2 + x_3^2x_1 + x_1^2x_2^3 + x_2^3x_2^3 \]
\[ y_3^1 = x_3^2 \] (14.7)

\[ y^2 = r_{Q_{299}}^{3,1}(x^3, x^1) = r_{Q_{299}}^{3,1}(x^3, x^3, x^3) = (x_0^3, x_1^3, x_2^3) = (y_0^2, y_1^2, y_2^2, y_3^2) \]
\[ y_0^2 = x_0^3 + x_1^3x_3^3 + x_3^3x_0 + x_0^3x_3 + x_1^3x_2^3 + x_3^3x_2^3 \]
\[ y_1^2 = x_1^3 + x_0^3x_3^3 + x_3^3x_1 + x_0^3x_3 + x_1^3x_2^3 + x_3^3x_2^3 + x_1^3x_3 + x_3^3x_1 + x_1^3x_3 \]
\[ y_2^2 = x_2^3 \]
\[ y_3^2 = x_3^3 \] (14.8)

\[ y^3 = r_{Q_{299}}^{1,2}(x^1, x^2) = r_{Q_{299}}^{2,2}(x^1, x^1, x^2, x^3) = (y_0^3, y_1^3, y_2^3, y_3^3) \]
\[ y_0^3 = x_0^1 + x_1^1x_3^1 + x_3^1x_0 + x_0^1x_3^1 + x_0^1x_3^1 + x_0^1x_3^1 \]
\[ y_1^3 = x_1^1 + x_2^1x_3^1 + x_3^1x_1 + x_1^1x_3^1 + x_1^1x_3^1 + x_1^1x_3^1 \]
\[ y_2^3 = x_2^1 + x_1^1x_3^1 + x_3^1x_2 + x_1^1x_3^1 + x_1^1x_3^1 + x_1^1x_3^1 \]
\[ y_3^3 = x_3^1 \] (14.9)
Bibliography


Bibliography


170


176
Bibliography


Werner Schindler and Andreas Wiemers. Efficient side-channel attacks on scalar blinding on elliptic curves with special structure. In *NIST Workshop on ECC Standards*, 2015. 62, 81


Gilbert S. Vernam. Secret signaling system, July 1919. US Patent 1,310,719. 20

Bibliography


List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Arithmetic Core</td>
</tr>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>ARX</td>
<td>Addition-Rotation-XOR</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit</td>
</tr>
<tr>
<td>ASIP</td>
<td>Application-Specific Instruction Processor</td>
</tr>
<tr>
<td>BMS</td>
<td>Block Memory Content Scrambling</td>
</tr>
<tr>
<td>BRAM</td>
<td>Block-RAM</td>
</tr>
<tr>
<td>CAESAR</td>
<td>Competition for Authenticated Encryption: Security, Applicability, and Robustness</td>
</tr>
<tr>
<td>CFGLUT</td>
<td>Configurable Look-Up Table</td>
</tr>
<tr>
<td>CFRG</td>
<td>Crypto Forum Research Group</td>
</tr>
<tr>
<td>CI</td>
<td>Correlation Immune</td>
</tr>
<tr>
<td>CLB</td>
<td>Configurable Logic Block</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary Metal-Oxide-Semiconductor</td>
</tr>
<tr>
<td>CPA</td>
<td>Correlation Power Analysis</td>
</tr>
<tr>
<td>CPRNG</td>
<td>Cryptographic Pseudo Random Number Generator</td>
</tr>
<tr>
<td>CPS</td>
<td>Cyber Physical System</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CTR</td>
<td>Counter Mode</td>
</tr>
<tr>
<td>DCA</td>
<td>Differential Computational Analysis</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard</td>
</tr>
<tr>
<td>DLOG</td>
<td>Discrete Logarithm Problem</td>
</tr>
<tr>
<td>DPA</td>
<td>Differential Power Analysis</td>
</tr>
<tr>
<td>DRM</td>
<td>Digital Rights Management</td>
</tr>
<tr>
<td>DSA</td>
<td>Digital Signature Algorithm</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>DUT</td>
<td>Device Under Test</td>
</tr>
<tr>
<td>ECC</td>
<td>Elliptic Curve Cryptography</td>
</tr>
<tr>
<td>EEA</td>
<td>Extended Euclidean Algorithm</td>
</tr>
<tr>
<td>EM</td>
<td>electromagnetic</td>
</tr>
<tr>
<td>FAU</td>
<td>Field Arithmetic Unit</td>
</tr>
<tr>
<td>FIA</td>
<td>Fault Injection Attack</td>
</tr>
<tr>
<td>FF</td>
<td>Flip-Flop</td>
</tr>
</tbody>
</table>
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>First In - First Out</td>
</tr>
<tr>
<td>FLT</td>
<td>Fermat’s Little Theorem</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>GE</td>
<td>Gate Equivalent</td>
</tr>
<tr>
<td>GF</td>
<td>Galois Field</td>
</tr>
<tr>
<td>HD</td>
<td>Hamming Distance</td>
</tr>
<tr>
<td>HO-TI</td>
<td>Higher-Order Threshold Implementation</td>
</tr>
<tr>
<td>HW</td>
<td>Hamming Weight</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>IRTF</td>
<td>Internet Research Task Force</td>
</tr>
<tr>
<td>ISR</td>
<td>Inverse Square Root</td>
</tr>
<tr>
<td>LFI</td>
<td>Laser Fault Injection</td>
</tr>
<tr>
<td>LFSR</td>
<td>Linear-Feedback Shift Register</td>
</tr>
<tr>
<td>LS</td>
<td>Logic Slice</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up Table</td>
</tr>
<tr>
<td>MIA</td>
<td>Mutual Information Analysis</td>
</tr>
<tr>
<td>µC</td>
<td>Microcontroller</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>OTP</td>
<td>One-Time Pad</td>
</tr>
<tr>
<td>PAR</td>
<td>place-and-route</td>
</tr>
<tr>
<td>PRNG</td>
<td>Pseudo Random Number Generator</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RC</td>
<td>Reduction Core</td>
</tr>
<tr>
<td>RE</td>
<td>Reverse Engineering</td>
</tr>
<tr>
<td>RFC</td>
<td>Request for Comments</td>
</tr>
<tr>
<td>RFID</td>
<td>Radio-Frequency Identification</td>
</tr>
<tr>
<td>RFT</td>
<td>Reconfigurable Function Table</td>
</tr>
<tr>
<td>RNG</td>
<td>Random Number Generator</td>
</tr>
<tr>
<td>ROM</td>
<td>Read-Only Memory</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest-Shamir-Adleman</td>
</tr>
<tr>
<td>SCA</td>
<td>Side-Channel Analysis</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>SLT</td>
<td>Substitution-Linear-Transformation</td>
</tr>
<tr>
<td>SM</td>
<td>Switch Matrix</td>
</tr>
<tr>
<td>SNR</td>
<td>Signal-to-Noise Ratio</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-a-Chip</td>
</tr>
<tr>
<td>SPA</td>
<td>Simple Power Analysis</td>
</tr>
<tr>
<td>SPN</td>
<td>Substitution-Permutation Network</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static Random Access Memory</td>
</tr>
<tr>
<td>SRL</td>
<td>Shift Register Look-Up Table</td>
</tr>
<tr>
<td>TI</td>
<td>Threshold Implementation</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
<tr>
<td>TRNG</td>
<td>True Random Number Generator</td>
</tr>
<tr>
<td>TVLA</td>
<td>Test Vector Leakage Assessment</td>
</tr>
</tbody>
</table>
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Schematic layout of a 6-input LUT with configurable shift register mode.</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Interface of a 36kb dual-port Block-RAM.</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Simplified architecture of a 7-series DSP48E1 slice.</td>
<td>14</td>
</tr>
<tr>
<td>5.1</td>
<td>Architecture overview of the basic AES-128 encryption core: The round function and key schedule are implemented in 15 slices, the remaining 6 slices are necessary for control logic (omitted in the figure for the sake of clarity).</td>
<td>43</td>
</tr>
<tr>
<td>5.2</td>
<td>Placement of the basic encryption architecture on a Spartan-6 FPGA (visualized using Xilinx PlanAhead tool).</td>
<td>46</td>
</tr>
<tr>
<td>5.3</td>
<td>Side-channel evaluation results for a CPA using the Hamming weight power model (for S-box input of last round without shuffling of the byte execution order).</td>
<td>49</td>
</tr>
<tr>
<td>5.4</td>
<td>Side-channel evaluation results for a CPA using the Hamming weight power model (for S-box input of last round with shuffling of the byte execution order).</td>
<td>49</td>
</tr>
<tr>
<td>5.5</td>
<td>Side-channel evaluation results for a CPA using the Hamming weight power model (for S-box input of last round without shuffling the byte execution order) with arithmetic noise.</td>
<td>50</td>
</tr>
<tr>
<td>5.6</td>
<td>Side-channel evaluation results for a CPA using the Hamming weight power model (for S-box input of last round with shuffling of the byte execution order) with arithmetic noise.</td>
<td>50</td>
</tr>
<tr>
<td>6.1</td>
<td>Visualization of the combined double-and-add formula according to Montgomery’s ladder.</td>
<td>56</td>
</tr>
<tr>
<td>6.2</td>
<td>Overview of the Curve25519 core.</td>
<td>58</td>
</tr>
<tr>
<td>6.3</td>
<td>Architecture of the modular multiplication unit.</td>
<td>59</td>
</tr>
<tr>
<td>6.4</td>
<td>Overview of the multi-core design.</td>
<td>60</td>
</tr>
<tr>
<td>6.5</td>
<td>Round-Robin-based load balancing scheme.</td>
<td>61</td>
</tr>
<tr>
<td>6.6</td>
<td>Concept of the random scalar blinding countermeasure.</td>
<td>63</td>
</tr>
<tr>
<td>6.7</td>
<td>Mean traces over 1000 measurements.</td>
<td>67</td>
</tr>
<tr>
<td>6.8</td>
<td>Non-specific t-test results with fixed base point: Profile 1 (10000 traces)</td>
<td>68</td>
</tr>
<tr>
<td>6.9</td>
<td>Non-specific t-test results with fixed base point: Profile 2 (100000 traces)</td>
<td>68</td>
</tr>
<tr>
<td>6.10</td>
<td>Non-specific t-test results with fixed base point: Profile 3 (100000 traces)</td>
<td>69</td>
</tr>
<tr>
<td>6.11</td>
<td>Non-specific t-test results with fixed base point: Profile 4 (1000000 traces)</td>
<td>69</td>
</tr>
<tr>
<td>6.12</td>
<td>Non-specific t-test results with fixed scalar: Profile 1 (100000 traces)</td>
<td>70</td>
</tr>
<tr>
<td>6.13</td>
<td>Non-specific t-test results with fixed scalar: Profile 2 (1000000 traces)</td>
<td>70</td>
</tr>
<tr>
<td>6.14</td>
<td>Non-specific t-test results with fixed scalar: Profile 3 (10000000 traces)</td>
<td>71</td>
</tr>
<tr>
<td>6.15</td>
<td>Non-specific t-test results with fixed scalar: Profile 4 (10000000 traces)</td>
<td>71</td>
</tr>
<tr>
<td>7.1</td>
<td>Custom schoolbook multiplication with interleaved reduction</td>
<td>78</td>
</tr>
<tr>
<td>7.2</td>
<td>Simplified architecture of the FAU</td>
<td>80</td>
</tr>
</tbody>
</table>
List of Figures

7.3 Mean traces over 1 000 measurements ........................................ 84
7.4 Non-specific t-test results with fixed base point: Profile 1 (10 000 traces) .... 85
7.5 Non-specific t-test results with fixed base point: Profile 2 (100 000 traces) ... 85
7.6 Non-specific t-test results with fixed base point: Profile 3 (100 000 traces) ... 86
7.7 Non-specific t-test results with fixed base point: Profile 4 (1 000 000 traces) ... 86
7.8 Non-specific t-test results with fixed scalar: Profile 1 (10 000 traces) .......... 87
7.9 Non-specific t-test results with fixed scalar: Profile 2 (100 000 traces) ........ 87
7.10 Non-specific t-test results with fixed scalar: Profile 2 (10 000 traces) ......... 88
7.11 Non-specific t-test results with fixed scalar: Profile 2 (1 000 000 traces) ....... 88

8.1 Structure of a CFGLUT and a (6 × 1) Reconfigurable Function Table .......... 93
8.2 Full-parallel round-based PRESENT encryption architecture .................... 94
8.3 4-bit PRESENT S-box using Reconfigurable Function Tables .................... 96
8.4 Sample Power Trace ...................................................................... 97
8.5 Profile 0 (No Countermeasure) with 1 000 000 traces ......................... 98
8.6 Profile 1 (Random S-box Decomposition) with 1 000 000 traces ............... 98
8.7 Profile 2 (Boolean Mabking) with 1 000 000 traces ............................ 99
8.8 Profile 3 (Register Precharge) with 1 000 000 traces ......................... 99
8.9 Profile 4 (Decomposition and Precharge) with 1 000 000 traces ......... 100
8.10 Profile 5 (Masking and Precharge) with 1 000 000 traces .................. 100
8.11 Profile 6 (Decomposition and Masking) with 1 000 000 traces ............ 101
8.12 Profile 7 (Decomposition, Masking, and Precharge) with 10 000 000 traces .................................................. 101

9.1 S-box architecture using RAM32M .................................................. 105
9.2 S-box architecture using RAM64M .................................................. 106
9.3 S-box architecture using RAM256X1S ............................................. 107
9.4 S-box architecture using RAMB8BWER .......................................... 108
9.5 Round-based AES implementation with Randomized Look-Up Tables .......... 109
9.6 First-order non-specific t-Test results .............................................. 112

10.1 A first-order TI of the PRESENT S-box ......................................... 115
10.2 Architecture of the PRESENT encryption design .............................. 119
10.3 Non-specific t-test results: Search ............................................... 122
10.4 Non-specific t-test results: Precompute ........................................... 123
10.5 Sample traces of a PRESENT encryption (Generate) .......................... 124
10.6 Non-specific t-test results: generate ............................................... 125

11.1 White-box implementation of a quarter AES round .......................... 132
11.2 A sample power tsrace ............................................................... 136
11.3 CPA results, S-box output bit model, 10 000 000 traces ...................... 138
11.4 Detailed representation of an 8 × 32 look-up table at the first round of our white-box design .................................................. 139
11.5 Walsh transforms for all 32 functions $f_k \in \{1, \ldots, 32\} (\cdot)$ with $HW(\omega) = 1$ .......................................................... 139
11.6 Walsh transforms for all 32 functions $f_k \in \{1, \ldots, 32\} (\cdot)$ with $HW(\omega) = 1$ for all key candidates $k^* \in \{0, 1\}^8$ ......................... 140
11.7 Sum of all imbalances $\Delta_k$ for all key candidates .......................... 141
12.1 First-order TI of the PRESENT S-box ........................................ 148
12.2 Quarter round of encoded PRESENT TI .................................... 149
12.3 Non-specific $t$-test results: Profile 1 (100 000 traces) .................. 152
12.4 Non-specific $t$-test results: Profile 2 (100 000 000 Traces) ............ 153
12.5 Non-specific $t$-test results: Profile 3 (1 500 000 000 traces) ........... 154
### List of Tables

5.1 Comparison of selected implementations of AES for FPGAs. .......................... 48
6.1 Area and delay overhead of countermeasures ............................................. 64
6.2 Summary of the device utilization and performance. ................................. 65
6.3 Selected high-performance implementations of ECC ................................. 66
7.1 Resource utilization and performance results after place-and-route ............. 82
7.2 Comparison of different designs for Elliptic Curve Cryptography over prime fields on FPGAs ................................................................. 83
8.1 Comparison of resource utilization and time overhead for unprotected and protected PRESENT designs .............................................................. 95
9.1 Comparison of S-boxes for different memory primitives ............................. 110
9.2 Time and resource requirements of entire AES (encryption only) ............... 111
10.1 The number of existing affine triples for different compositions ............... 118
10.2 Area and time overhead of different design approaches .......................... 121
11.1 Area and memory consumption of different table types ............................ 135
12.1 Area consumption of our hardware architecture .................................... 150
12.2 Comparison of different PRESENT Hardware Architectures .......................... 151
About the Author

Author information as of March 2017.

Personal Data

Name Pascal Sasdrich
Address Chair for Embedded Security, Ruhr-Universität Bochum, ID 2/637, Universitätsstr. 150, 44801 Bochum, Germany
E-Mail pascal.sasdrich@rub.de
Data of birth November 12, 1988
Place of birth Duisburg, Germany

Education

since 10/2013 Ruhr-Universität Bochum
Doctoral Candidate at Hardware Security Group/Chair for Embedded Security
Fast-Track PhD Candidate (TopING)

10/2012 – 03/2015 Ruhr-Universität Bochum
Master of Science (M.Sc.) in IT Security/Information Engineering
Average Score: Very Good (95%)

10/2009 – 09/2012 Ruhr-Universität Bochum
Bachelor of Science (B.Sc.) in IT Security/Information Engineering
Average Score: Very Good (89%)

Professional Experience

since 10/2013 Research Assistant
Ruhr-Universität Bochum, Hardware Security Group/Chair for Embedded Security

09/2017 – 11/2017 Electrical Engineering Intern
Cryptography Research Division, Rambus, San Francisco

04/2012 – 06/2012 Intern
ESCRYPT GmbH - Embedded Security, Bochum
Publications and Academic Activities

In the following, all formal and informal publications of the author of this thesis are listed (information as of October 2017). Additionally, we list invited talks, academic awards, and selected attended conferences, workshops, and summer schools.

Peer-Reviewed Journal Papers


Peer-Reviewed Conference Proceedings


Publications and Academic Activities


Technical Reports


Publications and Academic Activities


Invited Talks

- Implementing Curve25519 for Side-Channel-Protected Elliptic Curve Cryptography, *20th Workshop on Elliptic Curve Cryptography*, September 1-7, 2016, Yaşar University, İzmir, Turkey

Awards and Stipends

11/2013  CAST-Förderpreis IT-Sicherheit.
3rd price in category: Bachelor/Student Thesis.

since 10/2013  TopING Student.
Member of fast-track PhD programm (Ruhr-Universität Bochum).

10/2010 – 09/2014  Deutschlandstipendium
Scholarship for students who perform outstandingly academically and are involved in special social engagement.
Funded by the Horst Görtz Stiftung.
Participation in Selected Conferences, Workshops, and Summer Schools

- COSADE 2017 (Paris, France)
- RSA Conference Cryptographers’ Track (CT-RSA) 2017, (San Francisco, USA)
- Xilinx Security Working Group (XSWG) 2016 (Frankfurt, Germany)
- Final Conference on TRUDEVICE 2016 (Barcelona, Spain)
- ECC 2016 (İzmir, Turkey)
- CRYPTO 2016 (Santa Barbara, USA)
- CHES 2016 (Santa Barbara, USA)
- CrypTech Workshop (Berlin, Germany)
- ASAP 2016 (London, UK)
- FSE 2016 (Bochum, Germany)
- CHES 2015 (Saint Malo, France)
- SAC 2015 (Sackville, Canada)
- 22. Krypto-Tag 2015 (Munich, Germany)
- TI-Day 2015 (Leuven, Belgium)
- HOST 2015 (McLean, USA)
- COSADE 2015 (Berlin, Germany)
- Summer School on Real-World Crypto and Privacy 2014 (Sibenik, Croatia)
- 20. Krypto-Tag 2014 (Berlin, Germany)
- FPL 2014 (Munich, Germany)
- ARC 2014 (Vilamoura, Portugal)