Graz University of Technology
Institute for Applied Information Processing and Communications (IAIK)
Inffeldgasse 16a, 8010 Graz, Austria

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Integrated Systems Laboratory (IIS)
Gloriastrasse 35, 8092 Zurich, Switzerland

Faculty of Computer Science

Department of Information Technology and Electrical Engineering

Towards a DPA-Secure and High-Speed Authenticated Encryption System Based on Keccak

Master Thesis

Philipp Dunst
p.dunst@student.tugraz.at

May 2014

Supervisors: Dipl.-Ing. Michael Mühlberghuber, ETH Zurich, mbgh@iis.ee.ethz.ch
Dr. Frank K. Gürkaynak, ETH Zurich, kgf@ee.ethz.ch

Assessors: Dr. Hubert Kaeslin, ETH Zurich, kaeslin@ee.ethz.ch
Dr. Michael Hutter, TU Graz, michael.hutter@iaik.tugraz.at

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Acknowledgements

At first, I want to thank Michael Hutter from Graz University of Technology (TUG) and Michael Mühlberghuber from Swiss Federal Institute of Technology (ETH) Zurich for the excellent support during the last months. In particularly, I would like to thank Michael Hutter who also supervised other projects I worked on during my studies. In addition, I want to thank all members of the Microelectronics Design Center from ETH Zurich who helped me a lot designing my Application-Specific Integrated Circuit (ASIC). During my whole studies my colleagues Oliver Söll, Mario Werner, and Paul Wolfger were a great help. Thank you for the nice years. I would also like to thank my former school colleagues Jürgen Lecker, Markus Korbelius, Michael Maier, Stefan Ozlberger, and Lukas Walch. I am glad that we are still in contact. Last but not least I am deeply grateful to my whole family who supported me during the last years. Unfortunately, I missed the birth of my nephew, Matisse, because of the semester abroad. Therefore, I want to dedicate this work to him.

Abstract

Nowadays, cryptographic algorithms are widespread and used in different applications. Since the announcement that KECCAK is the Secure Hash Algorithm (SHA-3) competition winner, sponge-based algorithms get more and more important. Besides hashing, they can be used for encryption as well. Using encryption systems on wireless devices leads to two requirements on the implementation. First, wireless devices are often battery or externally powered. In order to increase the life time of the devices, low-area designs are necessary. Second, a public domain can access the device during computation. Attacks which focus on exploiting implementation properties of the device can use this opportunity to reduce the security of these devices. The most common attacks are based on power analyses which try to derive sensitive information from the power consumption during cryptographic computations. In order to prevent attacks on this level, countermeasures must be implemented. This leads to additional hardware components and therefore, increases the area of the whole implementation. The challenging task is to combine low-area designs with countermeasures against power analyses. Moreover, the increase of sensitive digital data imposes higher requirements on cryptographic implementations with regard to their throughput. In order to compensate this increase, speed-maximized implementations are necessary.

In this thesis, we present the implementation of an Authenticated Encryption (AE) system based on KECCAK. The goal was to design two different architectures. First, a low-area implementation of the system was aspired. This implementation is secured with countermeasures against power-analysis attacks. The techniques used are called *masking* and *hiding*. Three different masking schemes are implemented, resulting in three diverse instances all combined into an Application-Specific Integrated Circuit (ASIC), called ZORRO. Furthermore, a power-analysis attack against one instance is presented. Second, the goal was to maximize the throughput of the AE system. The architecture was implemented on a Field Programmable Gate Array (FPGA).

Keywords: Power analysis, KECCAK, secret sharing, masking, hiding, low-area, high-throughput, authenticated encryption

Kurzfassung

Verschlüsselungsalgorithmen sind weit verbreitet und werden in verschiedenen Anwendungen verwendet. Seit der Ankündigung, dass KECCAK den Sicheren-Hash-Algorithmus (SHA-3) Wettbewerb gewonnen hat, gewinnen Schwamm-basierte Algorithmen mehr und mehr an Bedeutung. Neben dem Einsatz als Hash-Funktion können jene Algorithmen auch für die Verschlüsselung verwendet werden. Verschlüsselungssysteme in drahtlosen Geräten führen zu zwei Anforderungen an deren Implementierungen. Erstens sind drahtlose Geräte oft batteriebetrieben oder extern versorgt. Um die Lebensdauer der Geräte zu erhöhen, sind kleinflächige Designs erforderlich. Zweitens sind sie einer Öffentlichkeit ausgesetzt welche auf die Geräte während der Berechnung zugreifen kann. Angriffe, die sich auf Implementierungseigenschaften konzentrieren, können diese Möglichkeit nutzen, um die Sicherheit dieser Geräte zu reduzieren. Die häufigsten Angriffe sind Leistungsanalysen. Jene Analysen versuchen Informationen aus dem Leistungsverbrauch, der während kryptographischer Berechnungen gemessen wurde, abzuleiten. Um Angriffe auf dieser Ebene zu verhindern, müssen Gegenmaßnahmen implementiert werden. Dies führt zu zusätzlichen Hardware-Komponenten und erhöht somit die Fläche der gesamten Implementierung. Eine anspruchsvolle Aufgabe ist es, kleinflächige Designs mit Gegenmaßnahmen gegen Leistungsanalysen zu kombinieren. Darüber hinaus, bringt der Anstieg von sensiblen digitalen Daten neue Herausforderungen an kryptographische Implementierungen mit sich. Um jenen Anstieg zu kompensieren ist eine Maximierung des Durchsatzes der Implementierungen erforderlich.

In dieser Arbeit präsentieren wir die Implementierung eines authentifizierten Verschlüsselungssystems basierend auf KECCAK. Das Ziel war es, zwei verschiedene Architekturen zu entwerfen. Zuerst wurde eine kleinflächige Implementierung des Systems angestrebt. Diese Implementierung ist zusätzlich mit Gegenmaßnahmen gegen Leistungsanalysen gesichert. Die verwendeten Techniken werden *Maskieren* und *Verstecken* genannt. Drei verschiedene Maskierungsschemen wurden implementiert. Alle wurden auf der resultierenden anwendungsspezifischen integrierten Schaltung, genannt ZORRO, kombiniert und fabriziert. Des Weiteren wird eine Leistungsanalyse einer Einheit präsentiert. Das zweite Ziel war es den Durchsatz des authentifizierten Verschlüsselungssystems zu maximieren. Die Architektur wurde auf einer Feld programmierbaren Gatter-Anordnung (FPGA) implementiert.

Stichwörter: Leistungsanalyse, KECCAK, Maskieren, Verstecken, kleinflächig, hoher Durchsatz, authentifizierte Verschlüsselung

Contents

List of Acronyms

AE	Authenticated Encryption
AES	Advanced Encryption Standard
ASIC	Application-Specific Integrated Circuit
ASMBL	Advanced Silicon Modular Block
CLB	Configurable Logic Block
CMOS	Complementary Metal Oxide Semiconductor
CPA	Correlation Power Analysis
DES	Data Encryption Standard
DRC	Design Rule Check
DSP	Digital Signal Processor
ECC	Elliptic Curve Cryptography
ETH	Swiss Federal Institute of Technology
FF	Flip-Flop
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GCM	Galois/Counter Mode
GE	Gate Equivalent

HM	Hiding Mode
I/O	Input/Output
IAIK	Institute for Applied Information Processing and Communications
IC	Integrated Circuit
IIS	Integrated Systems Laboratory
LFSR	Linear Feedback Shift Register
LUT	Lookup Table
LVS	Layout Versus Schematic
MAC	Message Authentication Code
MD	Message-Digest Algorithm
MM	Masked Mode
NIST	National Institute of Standards and Technology
NM	Normal Mode
OCB	Offset Codebook Mode
RAM	Random-Access Memory
ROM	Read-Only Memory
RSA	Rivest, Shamir, and Adleman
SCA	Side-Channel Attack
SHA	Secure Hash Algorithm
SMM	Secure Masked Mode
SPA	Simple Power Analysis
SS	Secret Sharing
TUG	Graz University of Technology
UMC	United Microelectronics Corporation
VCD	Value Change Dump
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Chapter 1

Introduction

Nowadays, wireless devices get more and more important in our world. These devices often operate on sensitive data. Cryptographic algorithms can be used to keep sensitive data secure. Recently, the National Institute of Standards and Technology (NIST) has announced competitions to find cryptographic algorithms which provide a high computational security. This security is very important, but wireless devices are also vulnerable in another point of view. They can be accessed by a public domain. Thus, the public domain has open access to the device during computation. This makes wireless devices vulnerable against attacks which focus on implementation facts. A common approach is to analyze the power consumption during computation. Since the power consumption of a device is correlated to the intermediate values of the executed algorithm, they can reveal sensitive information about the processed data. In order to guarantee the system's cryptographic security, countermeasures against power analyses must be implemented. This can increase the area occupation in a high dimension. In addition, wireless devices are often battery powered. In order to increase the lifetime, an area reduction of the implementation often helps. Another requirement of implementations is the increasing demand on large data sets in nowadays networks. The latter requirement can be compensated by improving the throughput of implementations.

In October 2012, NIST announced [27] that KECCAK [4, 6, 8] is the Secure Hash Algorithm (SHA-3) competition winner. KECCAK is based on the so-called *sponge construction* [7]. Sponge-based algorithms are very flexible and can be used for different applications, e.g., hashing and Authenticated Encryption (AE). But straight forward hardware implementations of KECCAK-based AE systems are vulnerable against power-analysis attacks. Thus, using unprotected implementations on wireless devices is not a suitable approach. In order to protect implementations against power analyses, a common approach is *masking*. Securing a hardware implementation with masking results in a power-analysis secure system, if the new functions, used by the implementation, follow three simple rules mentioned by Nikova et al. [29]. Different approaches following these rules have been published focusing on KECCAK [5, 11]. In addition, there exist other, mostly smaller, approaches to make power-analysis attacks more complicated [21]. They are called *hiding* and can be combined with masking as well. Their common drawback is that they increase the area occupation of an implementation. Therefore, if hiding and/or masking is used the question arises if such implementations can fulfill the requirements of a wireless device. Moreover, the amount of sensitive data processed by wireless increased steadily. Hence, throughput-trimmed implementations of cryptographic algorithms are often needed. Today, many block-cipher implementations are available which achieve this requirement. Sponge-based algorithms are relatively new approaches, but get very important because of the Secure Hash Algorithm (SHA)-3 nomination. Therefore, a question of interest is, if sponge-based algorithms can perform more efficiently and keep up with block-cipher implementations in terms of the achieved throughput.

The goal of this thesis was to design two implementations, each serving as an AE system based on Keccak. The first architecture is a low-area Application-Specific Integrated Circuit (ASIC) implementation which is called Zorro. The ASIC is secured with masking and hiding schemes against power-analysis attacks. They can be activated in a single and a combined way. The resulting ASIC combines three different masking schemes. All instances work independently from each other. Hence, an analysis of their power consumption can be made independently for each instance. In the second implementation, the AE system is maximized for throughput. A fully parallelized round function is used. The used permutation of Keccak works with 24 iterative rounds. Therefore, 8 implementations are possible only by unrolling the rounds. All high-throughput implementations were done on a Field Programmable Gate Array (FPGA).

Related work which implements Keccak-f[1600] in a power analysis secure way only focuses on hashing or on the single permutation. Our system is a fully working AE system. Additionally, it offers the opportunity to activate and deactivate the different countermeasures and is therefore a perfect tool to evaluate the different countermeasures. As a result, we can present that each implemented instance of our system is more than two times smaller than other published implementations available so far. Hence, it represents the smallest AE system based on Keccak-f[1600] mentioned so far which is protected against power analyses. In numbers, our smallest implementation needs 14000 Gate Equivalents (GEs), the second implementation needs 14500 GEs, and the biggest implementation needs 17000 GEs. For the ASIC realization a United Microelectronics Corporation (UMC) 180 nm technology was used. All instances work with a largest size of 1088 bits and provide a security level of 256 bits. The maximum frequency is given by 200 MHz. Each instance can encrypt 1 to 2 Mbit/s even though masking is applied. Besides, a power-analysis attack against one instance is presented. The power traces for the attack are based on simulation results. With this attack, we were able to reduce the security of a non-protected encryption from 256 down to 96 bits. The throughput-trimmed implementations do not provide countermeasures against power analyses. The targeted platforms for the high-throughput implementations are *Xilinx Kintex-7* and *Xilinx Virtex-7* FPGAs. The fastest implementation has a throughput of more than 34 Gbit/s. All implementations work with a block size of 1344 bits and have a security of 128 bits.

The remainder of the work is structured as follows. In Chapter 2, the theory behind the implementations is given. At first, basics of cryptography are presented followed by hardware relevant topics. In Chapter 3, the KECCAK permutation, the *sponge construction*, and the *duplex construction* are explained. Subsequently, Chapter 4 provides power-analysis attacks. We start with an explanation of Simple Power Analysis (SPA) and Correlation Power Analysis (CPA) followed by countermeasures against power analyses. Afterwards, masking and Secret Sharing (SS) are described in detail. In Chapter 5, the architecture behind ZORRO is explained. In Chapter 6, a power analysis of one of the three instances on ZORRO is given. Chapter 7 presents the high-throughput implementations. Finally, in Chapter 8, we conclude the work and outline future work.

Chapter 2

Selected Cryptography- and Hardware-Relevant Topics

Nowadays, everybody is in contact with cryptography even though most of the time without noticing it. The spectrum where cryptography is used is widespread and ranges from mobile phones to banking cards. In order to speed up cryptographic computations, dedicated hardware components are produced.

This chapter provides a brief introduction into cryptography- and hardware-related topics, relevant for this work. In Section 2.1, asymmetric-key and symmetric-key cryptography are outlined. Section 2.2 covers hash functions, followed by Section 2.3 about Message Authentication Codes (MACs). Subsequently, Section 2.4 explains Authenticated Encryption (AE) systems. Section 2.5 describes Linear Feedback Shift Registers (LFSRs). Basics of synchronous digital circuits are given in Section 2.6. In Section 2.7, a possibility of data exchange in multi-clocked environments is outlined. Finally, Section 2.8 explains the basics of Field Programmable Gate Arrays (FPGAs) using the example of the *Xilinx-7 Family*.

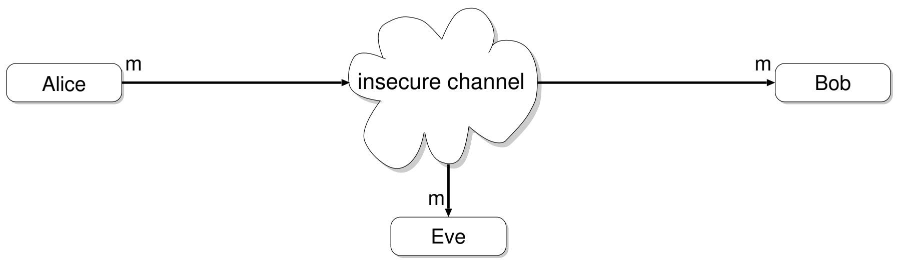


Figure 2.1.: A transmission of a message m from Alice to Bob over an insecure channel.

2.1. Why Cryptography?

Let Alice and Bob be two parties who want to exchange sensitive data, denoted by m (see Figure 2.1). They send their data over an insecure channel, such as the Internet. An evil person, called Eve, can easily read, alter, or even delete these data as long as she has access to the insecure channel. One of the applications of cryptography is to protect content from unauthorized reading. But there are more objectives that can be achieved with cryptographic algorithms. In literature, there exist four main objectives [22]:

- **Data Integrity:** A message which Alice sends to Bob cannot be manipulated by Eve without getting detected. Therefore, when Eve adds or changes data, Bob will notice it.
- **Confidentiality:** Only Alice and Bob can understand the content of the message the two parties exchange. Thus, a third party, for example Eve, cannot read the content.
- **Authentication:** Most of the time this aspect is separated in two parts. First, entity authentication which identifies the two communicating entities. Second, data-origin authentication which ensures that the message was sent actually by the expected party.
- **Non-repudiation:** When Alice sends a message, it can be proven that this message was sent actually by her. It allows Bob to prove that Alice has actually sent the message m even though Alice may claim something different at some later point in time.

There exist different reasons and ways of using cryptography. Sometimes not all objectives are needed. Therefore, many different primitives are available to achieve the wanted results.

2.1.1. Symmetric-Key vs. Asymmetric-Key Cryptography

Essentially, we can distinguish between symmetric-key and asymmetric-key cryptography. The main difference between symmetric-key and asymmetric-key cryptography is that in symmetric-key cryptography Alice and Bob have to exchange a common secret before they can start a cryptographic operation such as an en- or decryption (see Figure 2.2). Such a secret is usually called secret key, denoted by k. One big drawback of symmetric-key cryptography is the fact that k has to be distributed among the participating parties prior to the communication. If a third party gets access to the key, it can read the content of the messages, too. Therefore, keeping k secret is very important, but because of the required key exchange not always a simple task. The Advanced Encryption Standard (AES) and the Data Encryption Standard (DES) are very famous primitives based on symmetric-key cryptography.

In asymmetric-key cryptography (see Figure 2.3), every party gets a pair of keys, called private and public key. The public key can be accessed by everybody. An encryption can

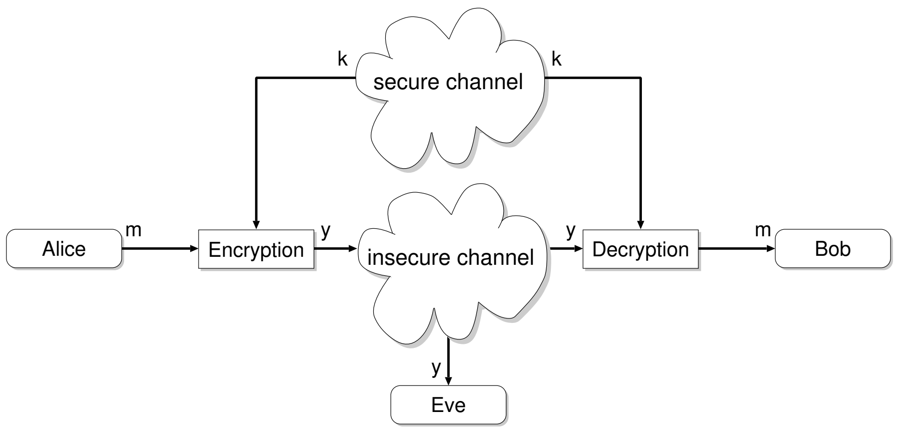


Figure 2.2.: Encrypted transmission of a message m from Alice to Bob over an insecure channel using symmetric-key cryptography.

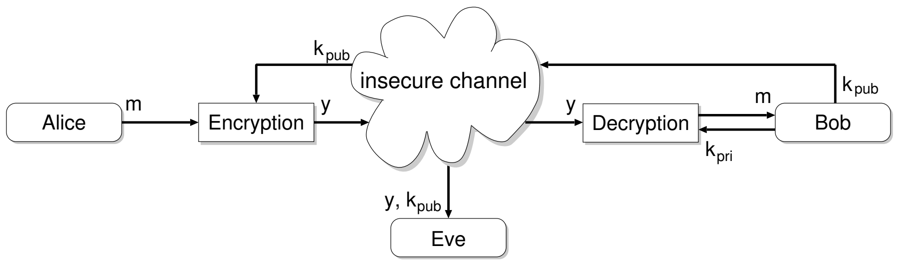


Figure 2.3.: Encrypted transmission of a message m from Alice to Bob over an insecure channel using asymmetric-key cryptography.

be made with this key and the resulting ciphertext can be sent to the owner of the key pair. Afterwards, the owner can decrypt the message with the private key. The private key belongs to the owner and therefore, must be kept secret. It is a big advantage that there is no need for a potentially vulnerable key exchange. Famous examples include the Rivest, Shamir, and Adleman (RSA) algorithm, which was called after their inventors, and Elliptic Curve Cryptography (ECC).

2.2. Hash Functions

When a message m is transmitted over an insecure channel, it is very often important to check the integrity of the message. Hash functions can be used to calculate a fixed-length digest from a message of arbitrary length. The digest then can be attached to the transmitted message in order to assure the message's integrity. After the transmission, the receiver can again hash the content. If the two hash values match, the message was not altered. Such a function does not need a key. But it must fulfill some general criteria to be secure. There exist three objectives in literature to describe the security of a hash function [31]:

- **Preimage resistance:** For a hash value y ($y = h(m)$, where h corresponds to a hash function) it should be infeasible to compute the input m of the function h. Or more precisely, a hash function should be a one way function. Therefore, it should not be possible to get the input with the help of the output.
- **Second preimage resistance:** It should be computationally infeasible to find for a given input m_1 another input m_2 with $m_1 \neq m_2$ that fulfills $y_1 = h(m_1) = h(m_2) = y_2$, where h corresponds to a hash function. In others words, it should be very difficult to find an input, different from a fixed one which leads to the same function output.
- **Collision resistance:** It should be computationally infeasible to find m_1 and m_2 with $m_1 \neq m_2$ that fulfills $y_1 = h(m_1) = h(m_2) = y_2$, where h corresponds to a hash function. The difference between this property and the second property is that more possibilities are available because m_1 is also freely selectable.

In order to find such functions, which ensure a high security, the National Institute of Standards and Technology (NIST) frequently announce competitions. In this way, the Secure Hash Algorithm (SHA) family occurred with the members SHA-1, SHA-2, and SHA-3 (not fully specified until now). Another famous function family is called Message-Digest Algorithm (MD). Up to now, this family has three famous members (MD-2, MD-4, and MD-5).

2.3. Message Authentication Codes

The basic idea of a Message Authentication Code (MAC) is to ensure that a message m is not altered during the transmission from Bob to Alice. In addition, such a scheme can

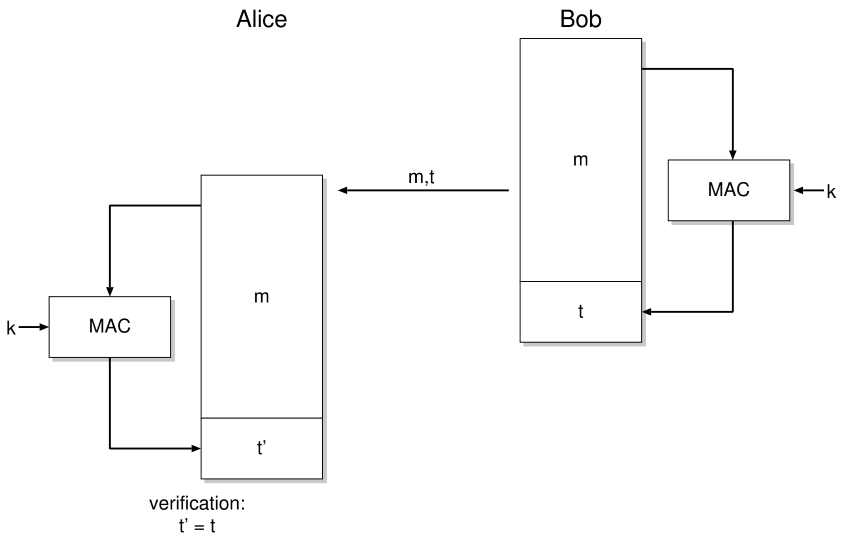


Figure 2.4.: MAC-based communication between Alice and Bob assuring integrity and authenticity.

guarantee authenticity. Therefore, Bob computes the MAC of the message with a shared secret key k. He transmits the MAC t attached to the original message to Alice. She also computes the MAC over m, again with k. If the computed and the received MAC match, the message is valid and m was not altered. Figure 2.4 shows the MAC computation and the exchange schematically. In practice, the MAC is shorter than m and has a fixed length like a hash function output. Because of the involvement of the symmetric key k, *data integrity* and *authentication* can be achieved. The following enumeration summarizes the achievements and properties of a MAC[31]:

- **Cryptographic checksum:** With a MAC you can create a cryptographic secure authentication tag for a given message.
- **Symmetric:** Because MACs are based on symmetric keys the two parties have to share a secret together.
- **Arbitrary message size:** A MAC must be able to handle messages of arbitrary length.
- **Fixed output length:** A MAC must have the same output length for each input.
- **Message integrity:** If the message changes, the MAC has to change, too. Therefore, it guarantees integrity.
- **Message authentication:** With a MAC the receiving party is assured of the origin of the message.
- **No non-repudiation:** Because MACs are based on symmetric principles, they do not provide *non-repudiation.*

2.4. Authenticated Encryption Schemes

As the name of the scheme already reveals, the system should be able to encrypt and provide authentication. In other words, it should provide objective one, two, and three mentioned in Section 2.1. To ensure these three objectives it needs [34]:

- **Encryption:**
  - Input: A plaintext, a key, and optionally a header
  - Output: A ciphertext and an authentication tag
- **Decryption**:
  - Input: A ciphertext, a key, and optionally a header
  - Output: A plaintext and an authentication tag

The key is used to ensure *confidentiality*. Therefore, only people who know the key can decrypt the data. The header does not get encrypted, but influences the authenticity protection. With the authentication tag, *data integrity* and *authentication* are ensured. An independent encryption system combined with a MAC computation can provide an Authenticated Encryption (AE) system. In addition, there exist modes which combine the encryption and the MAC computation. Most of the time the key is also used to ensure *authentication* in these modes. Examples for block-cipher based modes serving as AE system are the Offset Codebook Mode (OCB) and Galois/Counter Mode (GCM).

2.5. Linear Feedback Shift Registers

In cryptography, random numbers are often needed. The generation of real random values can be a hard task especially on hardware circuits. Another approach is to use so-called pseudo random values. To provide these pseudo random values, Linear Feedback Shift Registers (LFSRs) can be used.

2.5.1. The Structure of a Linear Feedback Shift Register

The main parts of an LFSR are clocked storage elements, for example Flip-Flops (FFs), and a feedback path. The number of storage elements defines the degree of the connection polynomial. In every clock cycle, the content of the storage elements is shifted to their lower partner (the FF next to it). Thus, one element does not get a new value from this mechanism. Therefore, a feedback path is needed. It sums up a predefined number of storage-element values and stores the result in the, not yet, updated element. The output is the content of the first element. In this way, one bit per cycle can be generated.

Consider m FFs and a switch s_i between their output and the feedback path. If the switch is activated, the value of the storage element is considered in the feedback path, otherwise it is not. All values of the considered elements get summed up. The sum

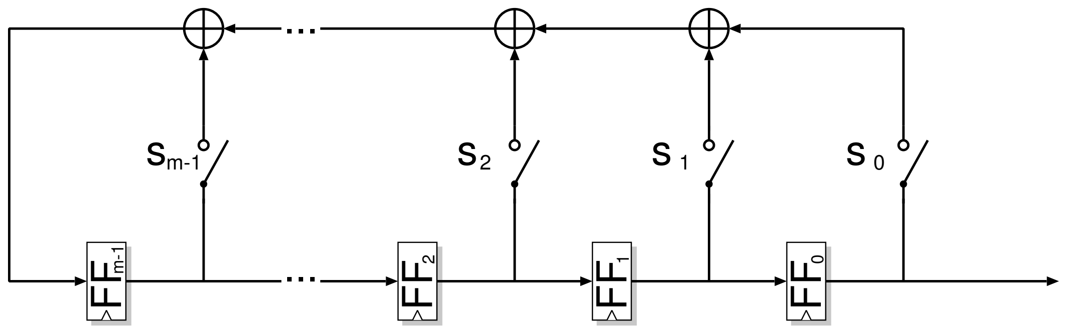


Figure 2.5.: General structure of an LFSR.

updates the highest $(m-1)$ element. With this more general description FF_{m-1} gets updated according to Equation (2.1). Figure 2.5 shows the basic structure of an LFSR.

$$FF_{m-1} = s_{m-1}FF_{m-1} + ... + s_1FF_1 + s_0FF_0 \bmod 2 \tag{2.1}$$

Due to the fact that there are only finite different values for the storage elements, the output of an LFSR repeats periodically. The maximum period is $2^m - 1$. But there are also LFSRs with a lower period. This can be attributed to the structure of the feedback path. The maximum period can be explained as follows. There exist only 2^m different conditions for the state. The state condition change is a deterministic process. For a given state condition, there is only one option for next state condition. Therefore, if a state condition is one time the same as a previous state condition, the LFSR starts to repeat. Additionally, if the state is zero, it stays zero. Therefore, this possibility must be excluded and the maximal period is given by $2^m - 1$. LFSRs with such a period are called maximum length LFSRs [31].

2.6. Synchronous Digital Circuits

When it comes to digital circuits, two main groups of elements are present. The first are combinatorial elements. They instantly change their output for new inputs. Examples for this group are XOR or AND operations. The other group consists of sequential elements. They are able to store information. Examples for this group are FFs and latches. A very famous approach to tell these elements to change their stored information is called synchronous clocking. In this approach, all elements change their information at a predefined periodical time. The point of time is defined by a so-called clock. The task of this signal is to generate a periodical wave that is connected to all sequential elements. Most of the time the waveform of a clock has the shape of a rectangle. Hence, there exist two activities that appear periodically. A positive and a negative edge. Therefore, these activities can be taken as a point of time to change the stored information. Consider a circuit that only uses FFs. Two different clocking approaches are possible. The first one is to make the FFs sensible only to the positive or negative edge of the clock. This category is called *single-edge-triggered one phase clocking.* The second one gives the FFs the chance to change their content twice in a cycle. They are able to change it at the positive and the negative edge. This category is called *dual-edge-triggered one phase clocking.* More details about different clocking schemes can be found in [17].

The distribution of the clock is a very complicated and important part in Integrated Circuit (IC) designs. In an optimal environment, the signal arrives at each storage element at the same time. In practice, this is not always possible because of different locations of the elements, different temperatures, or production variations. Hence, in real life we have to deal with different phenomenons [17]:

- **Clock skew:** Defines the various times of arrival of the clock for different elements for the same edge.
- **Clock jitter:** Defines the various times of arrival of the clock for the same element for different edges.

Most of the time the clock network is the biggest network on an IC. In order to guarantee that all sequential elements get updated in the correct way, the clock distribution is considered in a special.

The longest transmission time between sequential elements in a certain (sub)circuit defines the maximum frequency and is called *critical path.* Main processes in circuits are transmissions of data between several storage elements. All transmissions have to fulfill some timing behaviors to run off correctly. To describe them, four different delays are given in literature [17]:

- **Propagation delay (t_{pd}):** Defines the time a unit needs to react to a new input. After this delay all signals have settled in the unit and therefore the right output is provided.
- **Contamination delay (t_{cd}):** Defines the time a unit needs to react to a new input for the first time. Till now not all signals in the unit have settled. Hence, the output is not the desired one.
- **Setup time (t_{su}):** Defines the time the input of a unit is not allowed to change before a clock edge. This type of time is only needed for sequential elements.
- **Hold time (t_{ho}):** Defines the time the input of a unit is not allowed to change after a clock edge. This type of time is again only needed for sequential elements.

Consider two units, a producer who sends data to another unit, called the receiver, and the receiver himself. There exist two windows to explain the transmission between these two units. The first one is the *data-valid window.* In this window, the real data are valid on the receiver's input. This is the case after the propagation delay of the producer ($t_{pd_{pro}}$). The next window is called *data-call window.* It defines the time, the producer needs settled data on its input. So it has to fit in the *data-valid window*, otherwise the receiver updates its storage in a wrong way. An illustration of how the different delays and times play together can be found in Figure 2.6.

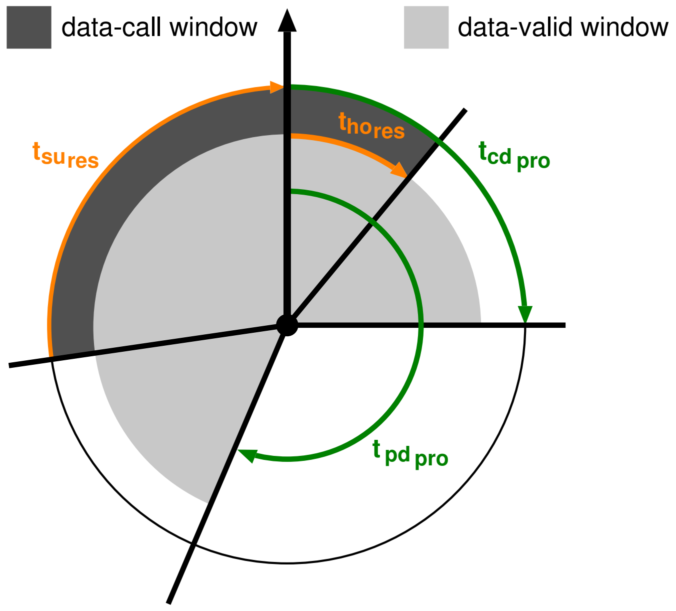


Figure 2.6.: Anceau diagramm showing the *data-valid* and the *data-call window*.

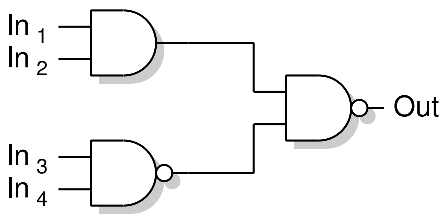


Figure 2.7.: Example of a fully combinatorial circuit consisting of AND and NAND gates.

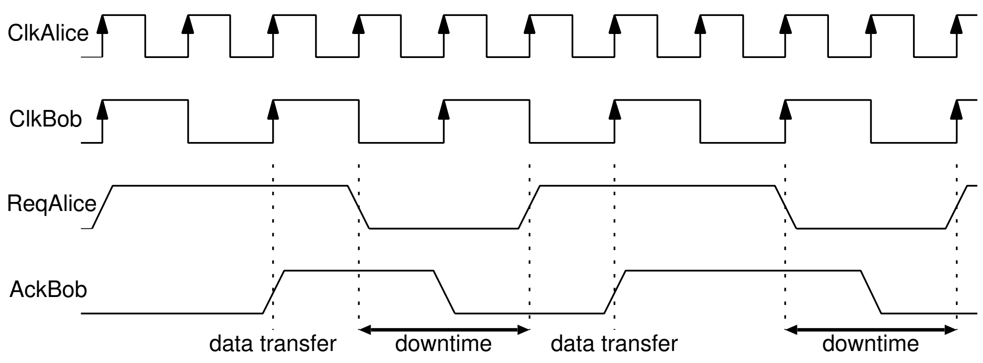


Figure 2.8.: Sequence diagram for sending data from Alice to Bob using a four way handshake protocol.

Nevertheless, there exist other events in digital circuits, too. Consider only a combinatorial circuit like in Figure 2.7. In an optimal environment, we consider that the output of this circuit only changes once in a cycle. But in real life, different logic elements have different propagation delays. The result is that the output can change more than once in the same activity cycle. These activities are called *glitches* and can play a big role if it comes to Side-Channel Attacks (SCAs).

2.7. Data Exchange in a Multi-Clocked Environment

Nowadays, systems often combine different independent hardware circuits. They are called system-on-chip designs. The various elements often have different timing behaviors and can therefore be clocked at varying maximum frequencies. A correct communication between them must be guaranteed and one way to do so is by utilizing handshake protocols. A subgroup is the so-called four way handshake protocol. Consider a producer, called Alice, and a consumer, called Bob. Alice wants to send data to Bob. Both work at several frequencies. Each party has a request and an acknowledge signal. When Alice wants to send new data, she applies the data on a shared data signal and raises the request signal. Now Bob can process the new data. Subsequently, he raises the acknowledge signal to show Alice that he does not need the data any longer. Afterwards, both signals must be zero again, first the request followed by the acknowledge signal. During this time, no new data can be exchanged. The protocol wastes this time of signal settlement. A sequence diagram of the two parties using a four way handshake protocol can be seen in Figure 2.8. The wasted time is called downtime in this diagram.

2.8. Basics of *Xilinx* Field Programmable Gate Arrays

Producing an Application-Specific Integrated Circuit (ASIC) is a task which needs a long time in comparison to software implementations. Hence, faster hardware would be desirable in order to reduce the time to market. A better solution can be achieved by using

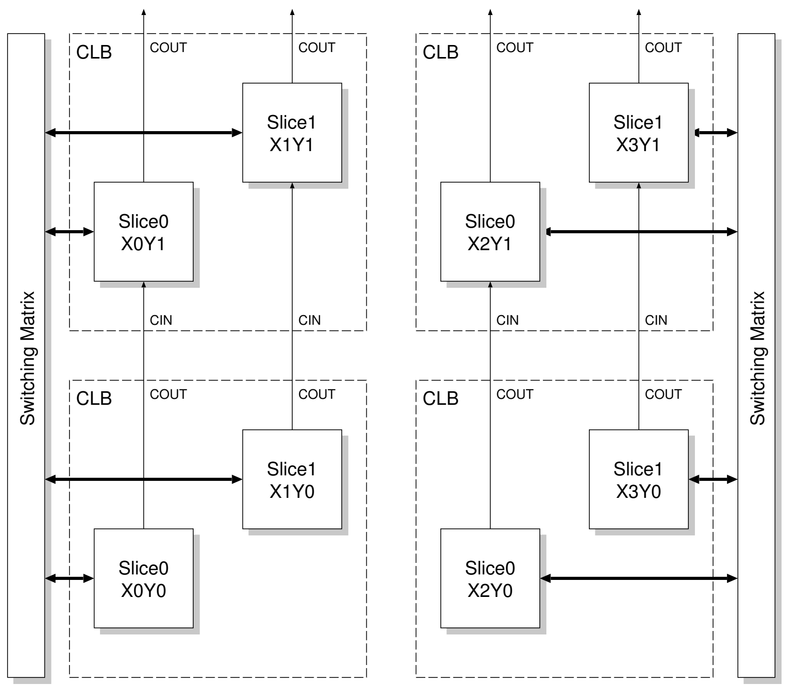


Figure 2.9.: Configurable Logic Block (CLB) organization in an FPGA of the *Xilinx-7 Family*.

FPGAs. As the name already reveals, these devices give the possibility of programmable hardware units. In comparison to ASICs, the maximum frequency decreases, but for small production quantities an FPGA approach can save costs. Hence, they can play an important role for hardware designs. This section gives an overview of how FPGAs work.

The structure is explained based on the *Xilinx-7 Family* which is based on the Advanced Silicon Modular Block (ASMBL) architecture. In order to get a flexible device, this architecture is column based. Each column represents a different type of block (for example memory, logic, Digital Signal Processor (DSP), or Input/Output (I/O) connections). All elements can be connected to a programmable switching matrix and can be combined in an arbitrary way.

2.8.1. Configurable Logic Blocks

One of the main blocks available on *Xilinx-7 Family* FPGAs are so-called Configurable Logic Block (CLB)s [42]. Each unit consists of two slices, called *Slice(1)* and *Slice(0)*. They are not directly connected. Additionally, the slices are organized as columns and each column shares a carry bit. To distinguish the different slices of several CLBs, x and y coordinates are available. They start with 0 and 0 on the bottom left corner of the die. Figure 2.9 shows the numeration for four CLBs. The *Xilinx-7 Family* provides two types of slices, called *SLICEL* and *SLICEM* (logic and memory). All slices provide

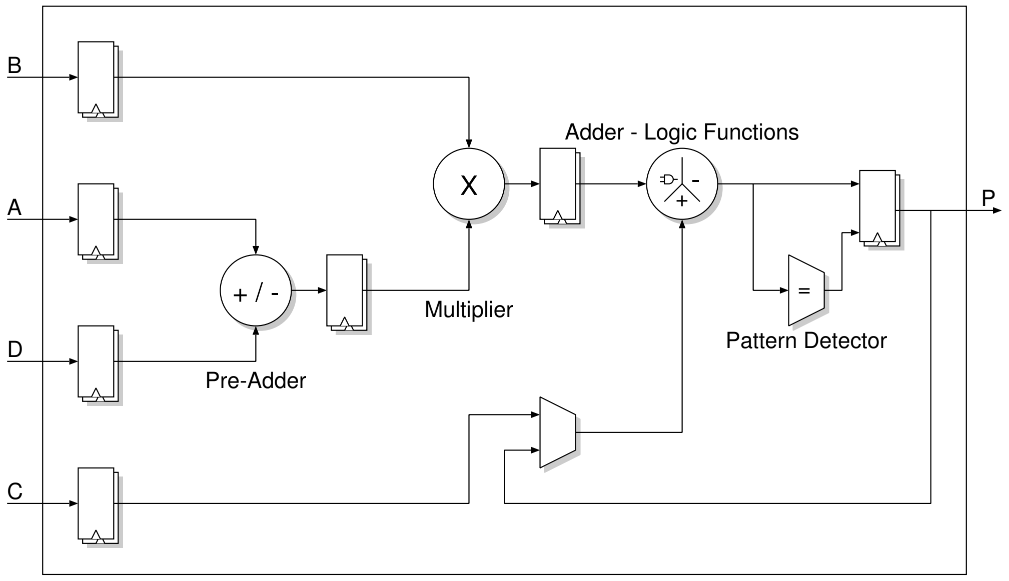


Figure 2.10.: *DSP48E1* unit provided in the *Xilinx-7 Family*.

four Look-up Tables (LUTs), eight storage elements, wide-function multiplexers, and a carry chain. The maximal size of the LUTs is given by 6 bits. They can be used to achieve combinatorial elements or can be configured as 64-bit Read-Only Memory (ROM). Hence, a slice can serve as 256-bit ROM. The storage elements can be used as FFs or latches. Additionally, *SLICEM* slices can be configured as 256-bit distributed Random-Access Memory (RAM) or as 128-bit shift register. The output of the LUTs can be directly transmitted to storage elements in the same slice, but not the other way around. Hence, a connection between a storage element and a LUT must be realized via the switching matrix which leads to an additional delay. With the combination of different LUTs arbitrary combinatorial circuits can be built.

2.8.2. Digital Signal Processors

The FPGAs of the *Xilinx-7 Family* also provide DSP blocks to speed up special operations. The instance used by Series-7 is called *DSP48E1* [43]. Each instance provides:

- 25×18 two's-complement multiplier
- 48-bit accumulator
- Power saving pre-adder
- Dual 24-bit or quad 12-bit add/subtract/accumulate
- Optional logic unit
- Pattern detector

Because of special pipeline registers and the hard wired elements, such blocks can speed up different types of computations. Especially, streaming operations can benefit from these pipeline structures. A multiply and accumulate computation is a very good example for a common function which can be executed with these blocks. An abstract overview of a *DSP48E1* unit can be seen in Figure 2.10.

2.8.3. Different Memory Types of the *Xilinx-7 Family*

There exist several memory types [45] in the *Xilinx-7 Family*, each focusing on different aspects.

Distributed RAM

As mentioned before, a *SLICEM* slice can be configured as RAM. The LUTs and storage elements are combined in a way to work as distributed RAM. Four different types are available:

- **Single Port:** One port for synchronous write and asynchronous read
- **Dual Port:** One port for synchronous write and asynchronous read and one for asynchronous read
- **Simple Dual Port:** One port for synchronous write and one for asynchronous read
- **Quad Port:** One port for synchronous write and asynchronous read and three for asynchronous read

The size of the RAMs varies from 256×1 to 32×1. But not all sizes are available for all types of configuration. Table 2.1 gives an overview of which sizes are available.

Shift Register

Shift registers are often required in hardware designs. In order to achieve them with a reasonable timing behavior *SLICEM* LUTs of the *Xilinx-7 Family* can be configured as 32-bit shift registers. If all four LUTs in a slice are cascaded, they can serve as a 128-bit shift register. For this configuration, the internal storage elements of the slices are not needed.

Block RAM

ICs often need a huge amount of memory. Therefore, many slices are needed as storage elements. In order to reduce the area for the memory and save the slices for combinatorial circuits, *Xilinx-7 Family* FPGAs also provide block RAMs. These units are separate independent units. Hence, they are connected via the switching matrices to the different other blocks. The delay of the additional routing often influences the timing behavior of the design significantly. Hence, a trade-off between time and area has to be made. In the

Table 2.1.: Available sizes of distributed RAM of the *Xilinx-7 FPGA Family*.

Size	**Configuration**
32×1	Single Port
32×1	Dual Port
32×2	Quad Port
32×3	Simple Dual Port
64×1	Single Port
64×1	Dual Port
64×1	Quad Port
64×3	Simple Dual Port
128×1	Single Port
128×1	Dual Port
256×1	Single Port

Table 2.2.: Different sizes of block RAM of the *Xilinx-7 FPGA Family*.

32 Kbit	**18 Kbit**
$32K \times 1$	$16K \times 1$
$16K \times 2$	$8K \times 2$
$8K \times 4$	$4K \times 4$
$4K \times 9$	$2K \times 9$
$2K \times 18$	$1K \times 18$
$1K \times 36$	512×36
512×72	

Xilinx-7 Family, each block RAM can store 36 Kbits and can be used as a single 36 Kbit block or as two 18 Kbit blocks. The different configurations can be seen in Table 2.2. The write and read operations are only available as synchronous simple dual port. The width of the read and write data signal can vary and must not be kept identical.

Chapter 3

The SHA-3 Competition Winner - KECCAK

In 2007, the National Institute of Standards and Technology (NIST) announced a public competition to develop a new cryptographic hash function. Sixty-four candidates participated [26]. In 2014, they announced [27] that KECCAK is the Secure Hash Algorithm (SHA)-3 competition winner.

The chapter focuses on KECCAK and is structured as follows. In Section 3.1, the naming convention and notation are outlined. Subsequently, Section 3.2 explains the different functions the permutation of KECCAK uses. Finally, Section 3.3 is about the *sponge construction*, the *duplex construction* and how an Authenticated Encryption (AE) system can be derived from them.

3.1. Naming Convention and Notation

The naming convention and notation for this chapter are the same as in the reference guide [4]. There exist 7 different KECCAK permutations. All of them differ from each other only because of the state size and the number of rounds. They are named after the following scheme: KECCAK-f[b], where $b = 25 \times 2^{\ell}$ and ℓ ranges from 0 to 6. Here b is the size of the state. The round quantity is defined as $12 + 2 \times \ell$. The state is arranged as a three dimensional array of $GF(2)$ elements with the structure $A[5]B[5]C[w]$, with $w = 2 \times \ell$. In order to convert the three dimensional array into a one-dimensional representation, the following equation can be applied: $S[w(5y + x) + z]$, with y and x modulo 5, and z modulo w. To distinguish between the different parts of the state, seven different elements exits:

- **Row:** A set of 5 bits with constant y and z coordinates.
- **Column:** A set of 5 bits with constant x and z coordinates.
- **Lane:** A set of w bits with constant x and y coordinates.

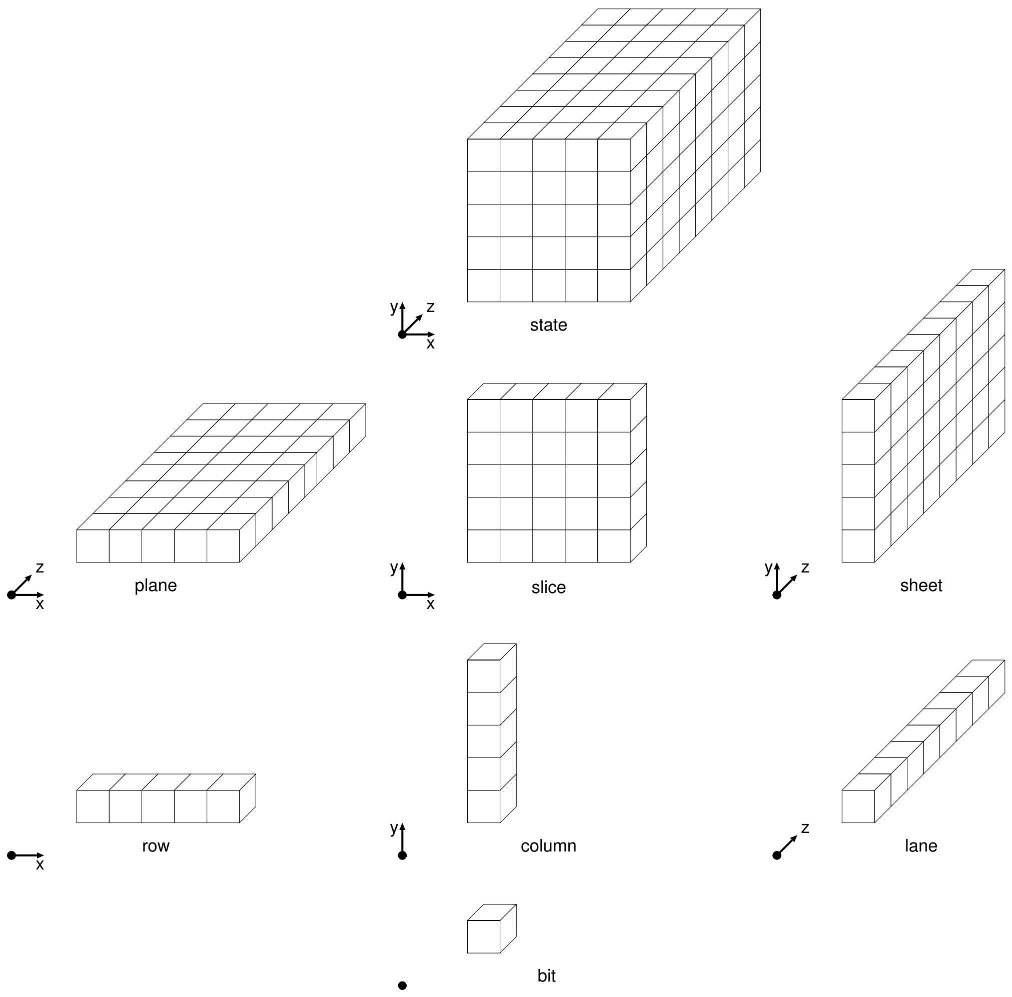


Figure 3.1.: Naming convention for the different elements of the KECCAK state.

- **Bit:** A set of 1 bit with constant x, y, and z coordinates.
- **Sheet:** A set of $5 \times w$ bits with constant x coordinate.
- **Plane:** A set of $5 \times w$ bits with constant y coordinate.
- **Slice:** A set of 25 bits with constant z coordinate.

Figure 3.1 shows the different parts of the KECCAK state. Notice that the bit $A[0][0][0]$ is not the bit in a corner of the array, it is the bit in the middle of the first slice.

3.2. The KECCAK Permutation

The heart of KECCAK is the permutation which can be divided into five different functions called θ, ρ, π, χ, and ι. Additionally, it has to be mentioned that KECCAK is an iterative

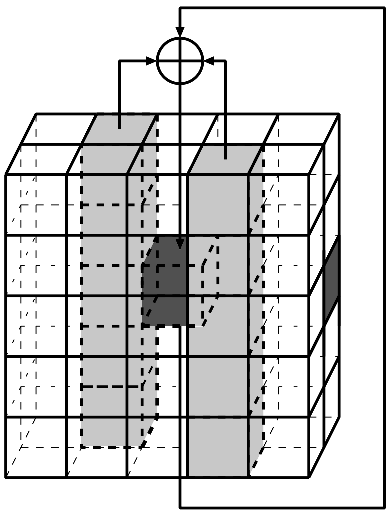


Figure 3.2.: Graphic view of the θ manipulating one bit of the state.

algorithm and all functions get executed on the whole state per round (except ι). A round R is defined as $R = \theta \times \rho \times \pi \times \chi \times \iota$.

3.2.1. θ - The Parity Function

This function works with the information of two slices at the same time. The mathematical description is given in Equation (3.1).

$$\theta : a[x][y][z] = a[x][y][z] + \sum_{y'=0}^{4} a[x-1][y'][z] + \sum_{y'=0}^{4} a[x+1][y'][z-1] \tag{3.1}$$

The result of each execution is the sum of 11 bits. The function is applied to each bit of the state. Figure 3.2 shows an illustration for θ manipulating one bit.

3.2.2. π - The Slice-Move Function

This function shifts all bits in a slice by a predefined constant according to Equation (3.2).

$$\pi : a[x][y] = a[x'][y'], \text{ with } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix} \tag{3.2}$$

Figure 3.3 illustrates the bit-movements within a single slice as defined by the π function. The only bit which is not altered is the first one ($x = 0$ and $y = 0$). This function is applied to each slice.

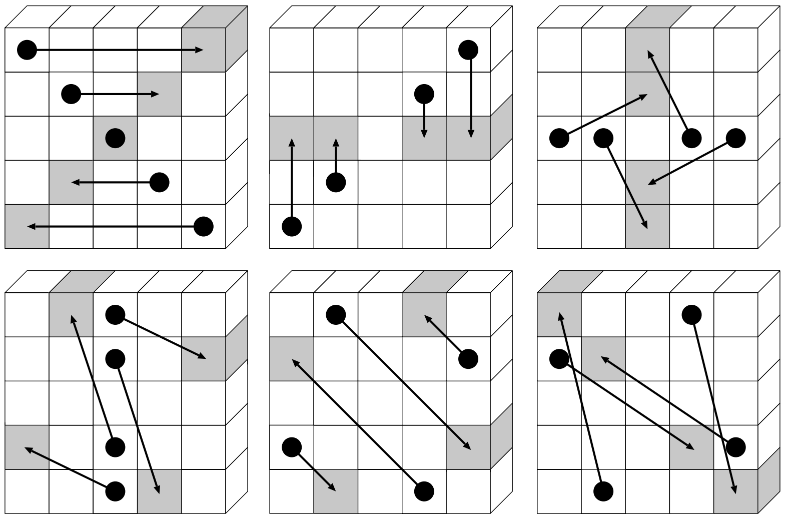


Figure 3.3.: The bit-movements performed by π on one slice.

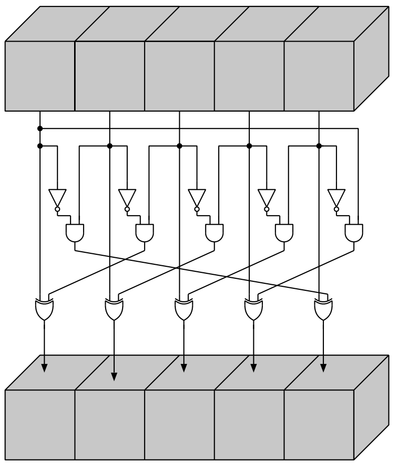


Figure 3.4.: Illustration for the χ function applied on one row.

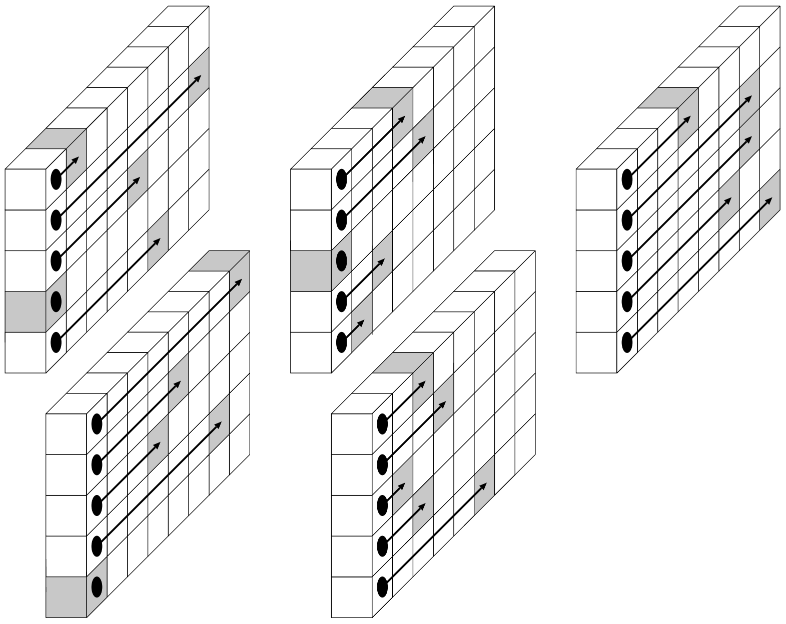


Figure 3.5.: Overview of the lane shift performed by ρ. w is assumed as 8.

3.2.3. χ - The Non-Linear Function

This operation works with one row at the same time. The computation follows Equation (3.3).

$$\chi : a[x] = a[x] + (a[x+1] + 1)a[x+2] \tag{3.3}$$

Because Keccak works in GF(2), all additions are XOR and the multiplications are AND operations. This function combines different bits of a row in a non-linear way. Figure 3.4 shows the non-linear combination χ performs on a row. It is applied to all rows of the state.

3.2.4. ρ - The Lane-Shift Function

This function works with the elements of the whole state and follows Equation (3.4).

$$\begin{gathered}\rho : a[x][y][z] = a[x][y][z - (t+1)(t+2)/2], \\ \text{with } t \text{ satisfying } 0 \leq t \leq 23 \text{ and } \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \text{ in } \mathrm{GF}(5)^{2\times 2}, \\ \text{or } t = -1 \text{ if } x = y = 0. \end{gathered} \tag{3.4}$$

Each lane gets shifted by a predefined constant. Table 3.1 shows the different constants for all lanes. They are valid for all instances of Keccak. To get the right value a modulo w operation must be applied to the constants. Figure 3.5 shows the graphical representation of this function with $w = 8$.

3.2.5. ι - Just a XOR Operation

This function only gets executed on the lane with $x = 0$ and $y = 0$. It is a very simple function which XORs a 64-bit constant to that lane. The mathematical description can be found in Equation (3.5).

$$\iota : a[0][0] = a[0][0] + RC[i_r] \tag{3.5}$$

The different round constants RC can be calculated with Equation (3.6) (the index i_r denotes to the round number):

$$RC[i_r][0][0][2^j - 1] = rc[j + 7i_r] \text{ for all } 0 \leq j \leq \ell. \tag{3.6}$$

The other values of RC are zero. The values for rc can be calculated with the help of the Linear Feedback Shift Register (LFSR) given in Equation (3.7).

$$rc[t] = (x^t \bmod x^8 + x^6 + x^5 + x^4 + 1) \bmod x \text{ in GF(2)[x]} \tag{3.7}$$

This function it the only one which differs in each round and is not executed on the whole state. To get an impression which values are used by ι, Table 3.2 shows the first five entries for KECCAK-*f*[1600] (notice that a KECCAK permutation with a smaller state size than 1600 can also take this table, but has to apply a modulo w operation to the table entries).

3.3. The *Sponge* and the *Duplex Construction*

A permutation alone is not that helpful. Hence, there exist sequences built around them to achieve different results, e.g., hash functions or encryption systems. In the case of KECCAK, this is the so-called *sponge construction*. Another approach is the *duplex construction*. This construction offers the possibility to build an AE scheme based on a permutation like the KECCAK-*f*[x] function.

Table 3.1.: Different shift values for θ.

	x=3	x=4	x=0	x=1	x=2
y=2	153	231	3	10	171
y=1	55	276	36	300	6
y=0	28	91	0	1	190
y=3	120	78	210	66	253
y=4	21	136	105	45	15

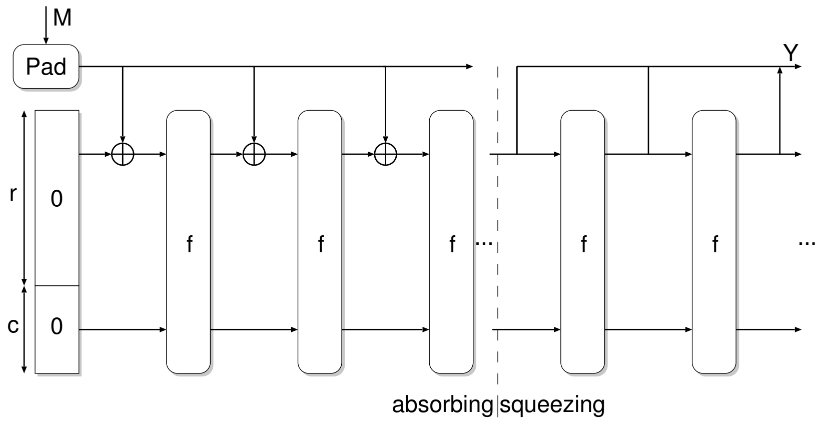


Figure 3.6.: *Sponge construction* used by KECCAK to hash messages.

3.3.1. The Sponge

The *sponge construction* is the most well know sequence. The resulting primitive is a hash function. It consists of two parameters called r, the bitrate, and c, the capacity and a permutation f. The input to the permutation f has the size $b = r + c$. The bitrate defines the input size the sponge can handle per permutation. The capacity is the rest of the state and is given by $c = b - r$. The authors of KECCAK defined the default size for KECCAK-f[1600] with $r = 1024$ bits and $c = 576$ bits. In addition, c defines the security level of the *sponge construction* which is given by $\frac{c}{2}$. In the case of KECCAK, the whole security of the primitive can be reduced to this expression.

The sequence consists of two phases. The *absorbing phase* and the *squeezing phase*. At first, the state gets initialized with zeros and the input, which should be hashed, is padded to a multiple of r. The KECCAK team uses the so-called $pad10 * 1$ rule. This scheme pads a single 1 followed by the minimum number of 0 followed by a 1 so that the size of the padded message is a multiple of r. Afterwards, the first phase starts. In this phase, the sponge XORs r bits of the message to the first r bits of the state, followed by a permutation which takes r and c as input. This is done until the whole input massage has been absorbed. Next, the second phase can start. Here the sponge outputs r bits of the state followed by a permutation. This step can be repeated until the user has

Table 3.2.: First five round constants for ι.

	$\mathbf{RC[i_r]}$
$i_r = 0$	$0x0000000000000001$
$i_r = 1$	$0x0000000000008082$
$i_r = 2$	$0x800000000000808a$
$i_r = 3$	$0x8000000080008000$
$i_r = 4$	$0x000000000000808b$

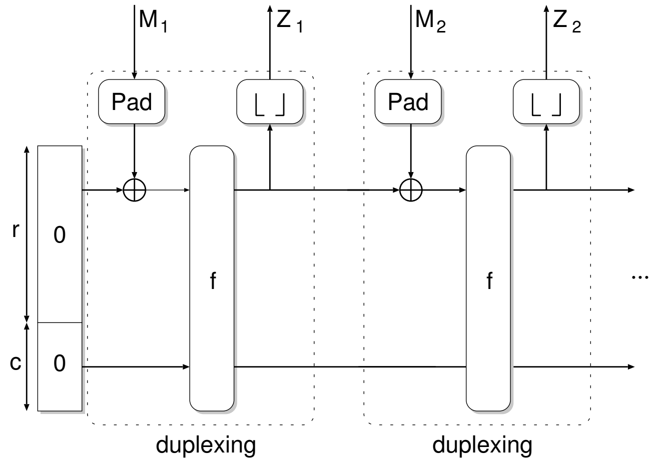


Figure 3.7.: *Duplex construction* [9].

the expected output length (notice that the output length can only be a multiple of r). Figure 3.6 shows the *sponge construction* graphically.

3.3.2. The Duplex

Another construction around the permutation is the so-called *duplex construction* [9]. Again there are the values r, c, and b which are related to $b = r + c$. This part is equal to the previous construction. The initialization of the state with zeros is also equal. In contrast to the *sponge construction*, the *duplex construction* consists of so-called *duplexing* entities. Such an entity needs two additional values, called σ and ℓ. σ is the input size the entity has to handle, ℓ is the output size. The next action, after the initialization, is to pad the input of the first *duplexing* unit to the size of r and XOR the padded input to the state. Then the permutation is performed. Afterwards, the entity outputs the first ℓ bits of the state, followed by another *duplexing* unit. The maximum size for σ and ℓ is r and the minimum size is 0. Up to n *duplexing* entities can be combined. Each entity can have different values for σ and ℓ. Figure 3.7 shows an illustration of the *duplex construction* using two *duplexing* entities.

3.3.3. How to build an Authenticated Encryption Scheme on the Base of the *Duplex Construction*

An AE system can be realized based on a *duplex construction.* First, n times $\sigma \geq 0$ bits can be handled as key and header bits. For those bits, ℓ can be kept to zero and therefore, no output is produced. After n permutations, the state of the construction is related to the key and the header bits. Afterwards, $\sigma = \ell \geq 1$ must be applied. Hence, each input bit leads to an output bit. Therefore, this can be the actual en-/decryption. It can be repeated up to o times (till all input bits are absorbed). For the authentication tag, m outputs from additional permutations can be taken. Now σ has to be zero and

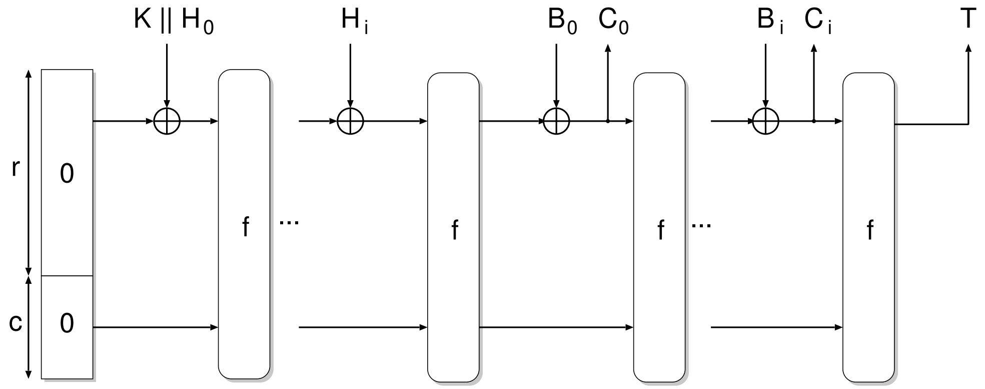


Figure 3.8.: Encryption of the plaintext B based on the *duplex construction*.

$\ell \geq 1$. Altogether the construction needs $n+o+m$ permutations for the computation. If the header, the key, or the input data cannot be applied to the construction because the size is not a multiple of σ, a padding scheme must be used. It is a great benefit of this scheme that the key size is not fixed. It can handle all lengths because of the padding. The security cannot be improved to an infinite level. Due to the fact that the security of the construction is $\frac{c}{2}$, a bigger key does not influence the security level, but it can handle them. So the construction is very flexible and it can share the key with other primitives even if the bit size does not match.

Nevertheless, let us have a more detailed view of how to get the ciphertext and the plaintext. After the key and the header are absorbed, the permutation gives us an output o for the first time. In encryption mode, this output can be combined with the first σ plaintext bits which are the input i_e of the user. Let us assume that this combination delivers the ciphertext c. It is defined as $c = i_e \oplus state[0:(\sigma-1)] = i_e \oplus o$. Afterwards, the first σ bits of the state are updated with c. This is a slight change compared to the original *duplex construction*. Now the structure processes first the input and afterwards outputs the bits. This can be repeated till all plaintext chunks are produced. The authentication tag is generated directly by the construction and it fulfills the requirements of a Message Authentication Code (MAC).

For the decryption, the combination of the bits is a little bit different. The input i_d of the user is now the ciphertext and not any longer the plaintext. But the state has to be updated as it was done for the encryption. Therefore, i_d has to influence the state directly. As an output of the construction, the XOR combination between the input of the user and the output of the permutation is taken again. The generation of the authentication tag remains unchanged.

Figure 3.8 and Figure 3.9 show the encryption and decryption scheme, respectively. Note that $\sigma = \ell = r$ is assumed for all phases. In addition, it is expected that the key, the header, and the input data are already padded. The output T (tag) corresponds to the MAC. If a MAC longer than r bits is needed, the last step can be repeated up to m times.

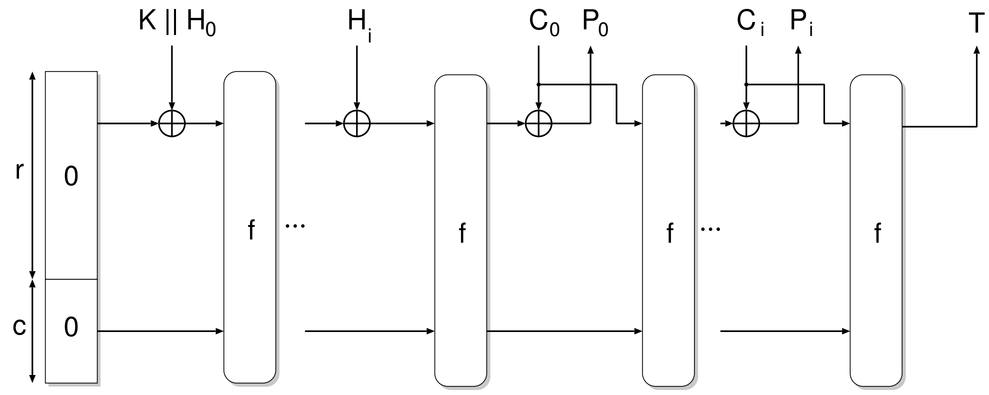


Figure 3.9.: Decryption of the ciphertext C based on the *duplex construction*.

Chapter 4

Power-Analysis Attacks

Today's cryptographic algorithms provide a very high level of security from a mathematical point of view. Most of the time, attacks on this level are computationally infeasible. But there exist other levels where attacks can be mounted. Power-analysis attacks investigate the power consumption of the device during computation. The main focus of this chapter is to introduce into power analysis attacks and possible countermeasures.

The chapter is structured as follows. In Section 4.1, the naming convention and notation are given. Section 4.2 explains the basics of power analysis. Afterwards, Section 4.3 gives a short introduction to Simple Power Analysis (SPA). In Section 4.4, Correlation Power Analysis (CPA) is described. Afterwards, countermeasures are presented. Section 4.5 starts with an explanation of hiding. Section 4.6 is about Secret Sharing (SS). First, the theory behind it is given, followed by different levels SS can be applied to. Subsequently, problems of a gate level masking scheme are given. Next, three different properties are explained, which must be fulfilled by an operation in order to be CPA secure. The last two subsections are about examples of SS schemes concerning Keccak and final notes on the countermeasures.

4.1. Naming Convention and Notation

Our reference literature to describe power-analysis attacks and countermeasures are Mangard et al. [21] and Nikova et al. [30]. When we discuss these issues, we try to keep the same conventions. A column vector is represented as a bold small letter like $\mathbf{v}$. A single element of it is represented as v_i. The size of the vector $\mathbf{v}$ is given by V. A matrix is represented as a big bold letter like $\mathbf{V}$. A matrix element is addressed with $v_{i,j}$, where i addresses the row and j the column. The size of a matrix is represented as $K \times R$. K is the number of rows, R is the number of columns.

4.2. Basics of Power Analysis

When it comes to a cryptographic algorithm, most of the time a secret key is involved. Therefore, to break such a system, it is necessary to get some information about the key or in the best case directly the plaintext. A powerful approach is to extract this information using power analysis. Nowadays, the mathematical security of cryptographic systems is very strong. Hence, such an attack is often the best way to break a system. The attack is based on the fact that the power consumption of a system depends on the data it processes. Thus, when the device processes instructions which work with the secret key, the power consumption can leak information about it. There are several ways to measure the power consumption of a device. The simplest way is to measure the current during computation. This can be made with a shunt resistor in the power supply line. For this method the attacker needs physical access to the device. Another way is the measurement of the electromagnetic field. Because the field is directly proportional to the power consumption, such a measurement can also be used for an attack. After the measurement, the analysis phase starts. In the literature, two basic attacks can be found. The first one focuses only on one trace. The other attack needs more power traces, but is more powerful. The two attacks are explained in the following sections.

4.3. Simple Power Analysis

For this attack only a single trace is needed. The goal is to have a closer look at the power trace when the key is involved during computation. When the exact time of this involvement can be found often the key can be revealed. Sometimes algorithms are round based. In other words, they use parts of key and compute the same operations iteratively. When the points of time of these operations can be found in the trace, they can be compared between each other. Sometimes this comparison can lead directly to key bits. An example of this attack can be shown when applied against an unprotected Rivest, Shamir, and Adleman (RSA) implementation. In order to speed up the computation, the exponentiation is often done with the help of the so-called *left-to-right Square and Multiply* algorithm. In this algorithm, depending on the current key bit, a square or a square and a multiply operation are performed. If the point of time of the execution of the *left-to-right Square and Multiply* algorithm can be found in the power trace, it can be possible to distinguish between a square and a multiply operation. This is possible because different operations lead to different power consumption forms. Therefore, information about the key can be gained from these different power consumption waveforms.

4.4. Correlation Power Analysis

This type of attack is very powerful because of the fact that the attacker does not need much information about the implementation of the algorithm. Most of the time knowing which algorithm gets performed by the device is enough. This knowledge can be gained through an SPA or even through product descriptions. On the other hand, most of the time a large number of traces is needed for a successful attack. Therefore, usually physical access to the device is necessary. Such an attack analyzes the power consumption at a fixed moment of time. It can be divided into five steps [21]:

- **Step one:** In this step, the attacker has to choose an intermediate value d of the algorithm which is computed by the device. This intermediate value has to depend on part of the secret key and a non-constant value. When it comes to a cryptographic function, most of the time this non-constant value is the plaintext or the ciphertext because the attacker can choose this value.

- **Step two:** In this step, the power traces, which the attacker needs, are measured. The attacker has to measure, the power trace for D different data blocks. During the measurement, the key part k, which is attacked, must be involved in the computation of the device. Additionally, the D different values are stored in a vector $\mathbf{d}= (d_i, ..., d_D)'$. For each chosen data block, the power traces correspond to $\mathbf{t_i'} = (t_{i,1}, ..., t_{i,T})$. Here, T denotes the sample quantity of the power trace. Thus, the measurement results can be written as a matrix $\mathbf{T}$ of the size $D \times T$. It is also important that the power traces are perfectly aligned. Therefore, it must be assured that during the measurement of $t_{y,1}$ and $t_{x,1}$, the exact same operation is performed.

- **Step Three:** In this step, the hypothetical intermediate values must by calculated. These are the different values for every possible choice of k and all D data blocks. Out of it, we get a vector $\mathbf{k} = k_1, ..., k_K$, where K denotes the total possible choices of k. Therefore, the attacker has to know which function is performed. With this knowledge it should be easy the get these values. This results in a matrix $\mathbf{V}$ of the size $D \times K$, where each row belongs to the same data block and each column to same key choice.

- **Step four:** In this step, the values from matrix $\mathbf{V}$ must be mapped to a power value. For this mapping there exist different power models. A power model is a function with an input and an output. The actual computation is a prediction of the power consumption because of the input. Therefore, it can take the hypothetical intermediate values as input and outputs hypothetical power consumption values. The most popular ones are the Hamming distance and the Hamming weight (see Section 4.4.1). But there are several other options [21]. Since the suitability of the applied power model depends on the actual design under test, different models should be considered. From this step, we get a matrix $\mathbf{H}$, again of the size $D \times V$, where each row belongs to the same data block and each column to the same key choice.

- **Step Five:** In this step, the comparison between the two matrices $\mathbf{H}$ and $\mathbf{T}$ is made. Each column $\mathbf{h_i}$ of matrix $\mathbf{H}$ is compared with each column $\mathbf{t_j}$ of matrix $\mathbf{T}$. Therefore, the attack compares the hypothetical power consumption values of each key with the recorded traces at every position. This results in a matrix $\mathbf{R}$ of the size $K \times T$, where each element $r_{i,j}$ contains the result of the comparison of $\mathbf{h_i}$ and $\mathbf{t_j}$. There are different methods to compare these two matrices, but in the case of a CPA a correlation is used. When all elements of $\mathbf{R}$ have nearly the same values the reason may be that not enough power traces have been measured.

The process of a CPA flow can be seen in Figure 4.1.

4.4.1. Power Models

In the last section, we explained how to perform a CPA. In step four, it comes to map the hypothetical intermediate values to power values. For a model which fits the power consumption in a better way, we can achieve an attack with a lower amount of traces. Therefore, it is very important to choose a good power model. As mentioned before, very popular models are the Hamming weight and the Hamming distance. Due to that fact, we want to spend more attention to these two methods (for other mapping types, we want to refer to [21]).

- **Hamming weight:** The Hamming weight of a string is the number of the symbols that are not 0 (see Figure 4.2). In other words, if a string is provided in binary representation it is the quantity of 1*s* in this string. This model can achieve good results because of the fact that writing a 1 into a storage element needs normally more power than writing a 0.

- **Hamming distance:** The Hamming distance is the sum of the symbols that are different in two strings (see Figure 4.2). Therefore, if it comes to a binary expression of two strings, we can calculate the distance with a XOR operation of these two values. It sums up the number of bits which are different.

These two types are very well known when register values are attacked. Additionally, they are very easy models because the attacker does not need a lot of information about the implementation to calculate them. Today, such models are often not good enough for attacks on more particular areas than register values. Therefore, often more sophisticated models have to be found. But in many cases, these models are very difficult to find because a lot of detailed knowledge of the attacked device has to be available. This can make attacks with more sophisticated models very hard.

4.5. Hiding

Because of the power and also simplicity of such attacks, several countermeasures were developed. One of them is called *hiding*. CPA attacks work because of the fact that the power consumption is correlated to the intermediate values processed by the design under test. Hiding tries to decouple this relation with two different approaches.

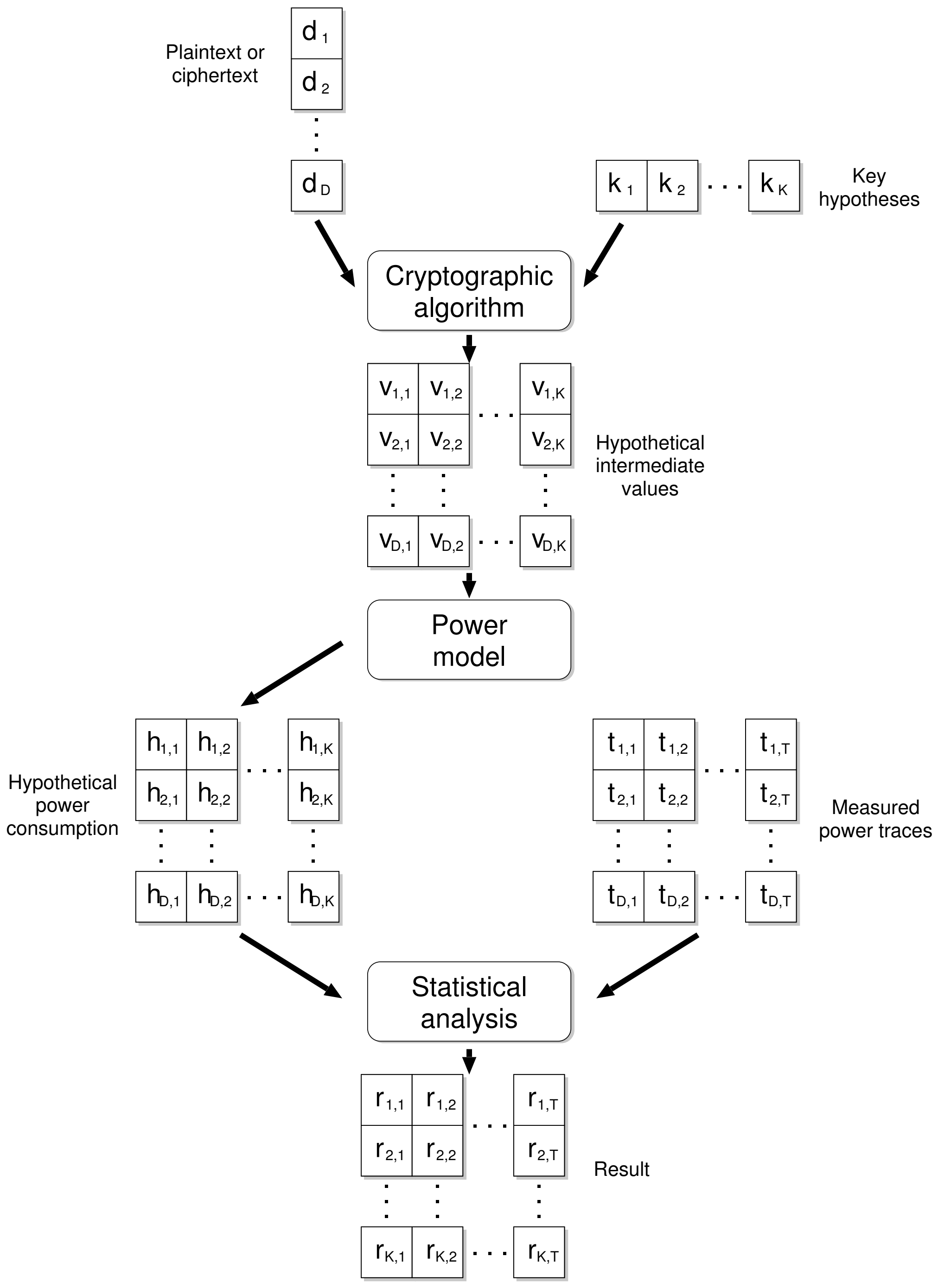

Plaintext or ciphertext
d_1
d_2
d_D
k_1 k_2 ... k_K
Key hypotheses
Cryptographic algorithm
$v_{1,1}$ $v_{1,2}$ ... $v_{1,K}$
$v_{2,1}$ $v_{2,2}$ ... $v_{2,K}$
$v_{D,1}$ $v_{D,2}$ ... $v_{D,K}$
Hypothetical intermediate values
Power model
Hypothetical power consumption
$h_{1,1}$ $h_{1,2}$ ... $h_{1,K}$
$h_{2,1}$ $h_{2,2}$ ... $h_{2,K}$
$h_{D,1}$ $h_{D,2}$ ... $h_{D,K}$
$t_{1,1}$ $t_{1,2}$ ... $t_{1,T}$
$t_{2,1}$ $t_{2,2}$ ... $t_{2,T}$
$t_{D,1}$ $t_{D,2}$ ... $t_{D,T}$
Measured power traces
Statistical analysis
$r_{1,1}$ $r_{1,2}$ ... $r_{1,T}$
$r_{2,1}$ $r_{2,2}$ ... $r_{2,T}$
$r_{K,1}$ $r_{K,2}$ ... $r_{K,T}$
Result

Figure 4.1.: Illustration of the five steps of a power analysis.

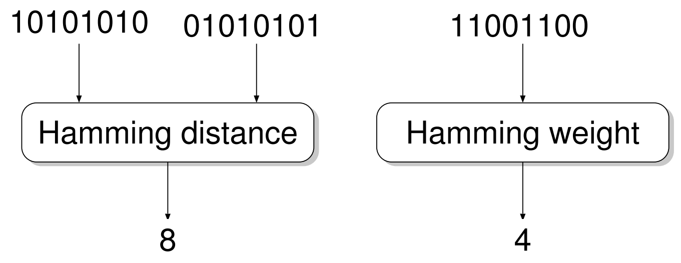


Figure 4.2.: The Hamming weight and the Hamming distance for 8-bit values.

4.5.1. Equalizing the Power Consumption

This form of hiding tries to decouple the power consumption and the values which get processed. This is done by an equalization of the power consumption. Therefore, each operation must consume the same power for each possible input value. There are many different approaches to achieve this. One operates on the gate level. Each gate must be built in a way so that the power consumption does not give information about the values it processes. Another approach is to duplicate all instances. The duplicated instances perform the operations in an inverted way. This should equalize the power consumption. In theory, this works very well, but in practice such an attempt is very complex and cannot be achieved exactly because to many variables have to be considered. Problems can arise because of production variations which lead to asymmetry. The number of needed traces increases most of the time in a quadratic way with these approaches. Hence, they are nice features, but an attack can still be successful.

4.5.2. Randomizing the Power Consumption

Another approach is to randomize the power consumption. Therefore, dummy operations are inserted. Such operations are performed before, during, or after the real operations. They do not operate on the real intermediate values and cannot be connected to the secret. The moment when a dummy operation is performed is randomly chosen for each execution of the algorithm. An arbitrary number of dummy operation can be inserted. It is a drawback that the execution time increases because of these additional computations.

A different way of randomization is to shuffle the operations of a device. For example, if there are operations that do not depend on each other, the execution order is not important. If these operations get exchanged, the result will be the same, but the point of time of the computation varies. Although it is not possible to avoid computing with the real intermediate values, the computation time does not increase. The fact that not all algorithms are suitable for such an approach is a drawback.

Nevertheless, securing implementations with hiding squares the needed trace for a successful power-analysis attack. A complete prohibition of CPA attacks is again not possible and therefore more powerful methods must be applied.

4.6. Masking

Very famous and more powerful countermeasures are called masking. This countermeasure masks the real processed intermediate value. Another, but very similar countermeasure technique is called Secret Sharing (SS). SS splits intermediate values in so-called shares. In this section, we want to introduce SS in detail.

4.6.1. The Theory Behind Secret Sharing

CPA uses the fact that there exists a correlation between the power consumption and the processed intermediate values of an algorithm. A SS scheme tries to decouple this relation. A total unlink between the intermediate values cannot be achieved without changing them. The main idea is to split the processed values into n shares. Let the number of shares be equal to two. Thus, from now on, we have two values (s_1 and s_2) which represent our intermediate value r. There must be a mathematical function which connects s_1 and s_2 to r. This is done with $r = s_1 * s_2$ or $s_2 = r * s_1$. In others words, r gets masked with the value s_1, which also explains the name. We call s_2 and s_1 protection value and protected value, respectively. The $*$ operation depends on the operations that are used by the cryptographic algorithm. Usually, the Boolean exclusive-or function, the modular addition, or the modular multiplication are used for $*$.

Most of the time s_1 is directly applied on the plaintext or the key. If the algorithm uses non-linear functions, $A(r * s_1) * s_1 \neq A(r)$, where A corresponds to the algorithm, is given. Therefore, the ciphertext is different for a computation without SS. The solution is to consider s_2 by performing the algorithm on s_2 and s_1. After the computation, a combination of s_1 and s_2 should achieve in the desired result. In a mathematical view, the algorithm has to be changed such that $A_{old}(r) = A_{new}(s_1) * A_{new}(s_2)$, where A corresponds to the algorithm. Moreover, a very important property of s_1 is the fact that it must be random and a new value each time the algorithm gets executed. Generally spoken, if an implementation works with n shares, $n - 1$ shares have to be randomly chosen for each execution. Because the algorithm has to compute the result for all shares, the execution time and/or the area of the implementation increases.

Now let us assume that the combinational function is a Boolean exclusive-or operation. Equation (4.1) describes the system.

$$s_2 = r \oplus s_1 \tag{4.1}$$

The truth table of Equation (4.1) is shown in Table 4.1. The result s_2 is uniformly distributed and if the result is 1, r can be 1 or 0. The same is valid for $r = 0$. Notice that s_1 differs for each computation of the algorithm. Therefore, s_2 and s_1 do not reveal any information about r. Therefore, such a system is secure against CPA if the hardware circumstances are expected in an optimal way.

In the previous example, we worked with the Boolean exclusive-or operation. If the algorithm works with arithmetic functions, the scheme must use them, too. Sometimes algorithms use both, Boolean and arithmetic functions. In this case, the scheme must switch between them. All statements mentioned before can be used for to schemes with up to n shares.

Most of the time cryptographic algorithms combine linear and non-linear functions. A linear function f has the property $f(x * y) = f(x) * f(y)$. This group of functions is no problem for a sharing scheme because f can be applied to s_1 and s_2. A problem appears when it comes to non-linear functions because they have the property $f(x*y) \neq f(x) * f(y)$. Therefore, new mathematical descriptions must be found for non-linear functions.

4.6.2. Different Levels of Secret Sharing

The previous section gives a basic view of how SS works. In this section, we want to discuss the different levels at which such schemes can be installed. The two main levels of SS will be discussed.

Algorithm Level

At this level, the whole algorithm is adapted to perform the computation in a way so that the result is valid. Non-linear functions must be taken care of in a special way. This often results in functions which vary for each share and so the hardware overhead increases. Most of the time such an approach is very simple because the problem is faced at a very high level of abstraction. Other advantages are that such a scheme works for hardware and software implementations and most of the time it is adequate to split and combine the shares at the beginning and at the end of the algorithm. On the other hand, it is very easy to lose performance. Nevertheless, SS on the algorithm level is a widespread method of secure systems.

Gate Level

At this level, each used cell gets changed. This approach is not suitable for software implementations, but it is a very common way to protect hardware against CPA. Figure 4.3 shows a block diagram of a gate with and without protection. Here a_r, b_r, and q_r correspond to protection values and s_a, s_b and s_q correspond to protected values. Therefore, each input of the cell gets protected. A big advantage of SS on this level is

Table 4.1.: The truth table for a Boolean sharing scheme of the degree two. As combinational function a Boolean exclusive-or operation is used.

r	$\mathbf{s_1}$	$\mathbf{s_2}$
0	1	1
0	0	0
1	1	0
1	0	1

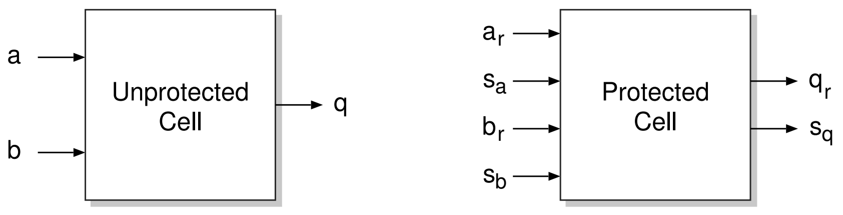


Figure 4.3.: Block diagram of a protected and a non-protected cell.

the fact that the algorithm itself stays the same. Additionally, the different cells can be used for several implementations.

How many different protection values are used is an important question. In literature, you can find three different approaches. The first one is to use a different random value for every input. This approach leads to an extremely huge amount of random values. Therefore, this approach is not practically usable.

The second one is to divide the different signals into so-called groups. Each group has the same random value. Now the amount of random values is smaller. A negative aspect of this method is the fact that additional logic is needed to manage the transfer of a signal from one to another group.

The third and last approach is to use the same value for each protection value. If two values depend on each other, they also depend on each other after applying the protection value. Most of the time a change of the random value can be detected in the power consumption because the network of this value has to be connected to each cell. Thus, a change has a big influence on the power consumption. This is another drawback of this solution.

4.6.3. Theory vs. Reality of Secret Sharing

In the previous sections, we explained SS. We only discussed the mathematical description of such a system and ignored the physical aspects. In this section, we focus on gate level SS and have a detailed look at glitches. A glitch occurs if a signal changes its value more than once during the same activity cycle. This can happen very easily because of different propagation delays of signals and is a normal thing to happen in Integrated Circuit (IC)s. Therefore, it is very important to analyze this effect and find out if it is a problem when it occurs in protected cells. Figure 4.3 shows a block diagram of such a cell. Now we want to have a closer look at the inner life of it. Figure 4.4 shows the structure of an AND gate which is protected [40]. The gate has five inputs. The two random values m_x, m_y, the two shared inputs $x_m = x \oplus m_x$, $y_m = m_y \oplus y$, and a new random value m_z. The outputs are the random value m_z and the shared result z_m which is computed as shown in Equation (4.2).

$$z_m = x_m y_m \oplus (m_y x_m \oplus (m_x y_m \oplus (m_x m_y \oplus m_z))) \tag{4.2}$$

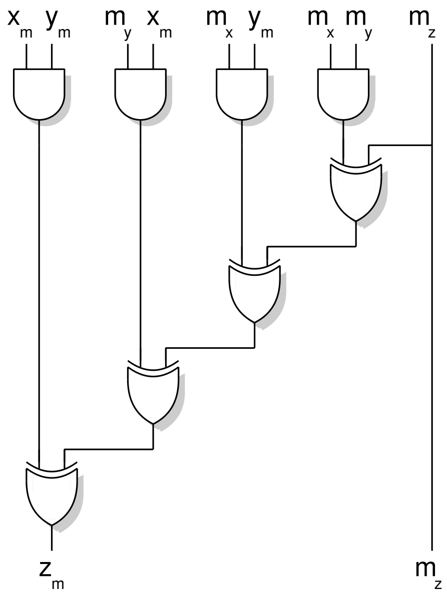


Figure 4.4.: Inner life of a protected AND cell [40].

The order of the XOR operations is not arbitrary. If the sum of one product is computed by the cell it can leak information of an intermediate value. For example, consider the first XOR. If at any time, it computes the sum of $x_m y_m$ and $m_y x_m$, information about y is leaked because $x_m y_m \oplus m_y x_m = y$. To get rid of this problem the new random value m_z was introduced.

Now let us assume that there is a glitch in the signal x_m. Because this value is the input of the first two AND gates it will influence them. Thus, a glitch on x_m has an influence on the values y_m and m_y. Let us have a look at the gates that change their output because of the glitch and the different values for y, m_y, and y_m. Table 4.2 shows the different numbers of gates which have been influenced. Consider that y has the value 1. Therefore, the mean influenced gate number is 2.5, for $y = 0$ we get 2.0. The number of gates that are influenced depends on the real intermediate value y. Therefore, such a gate can be attacked although it should be protected against CPA. There remains the problem that protected values, like y_m, and protection values, like m_y, get processed at the same time.

In order to show that not only a glitch in x_m can be dangerous, let us assume that m_x is now the targeted value. Table 4.3 shows the number of influenced gates. Again, there is a difference between the number for $y = 0$ (mean influenced gate number is 3) and for $y = 0$ (mean influenced gate number is 4.5). Glitches play a very important part for CPA and can therefore, not be ignored. Thus, we have to keep them in mind when designing circuits which should be secure against CPA.

4.6.4. How to Make Secret Sharing Secure

In the previous sections, we explained how SS works and why the first approach is not really secure when it comes to hardware implementations. In this section, we try to give an overview how to make a system secure against CPA.
SS is a very important and discussed topic and many people have worked on it. Many publications can be found focusing on different aspects. One of them is [30]. They defined three different properties a SS scheme must fulfill to become secure. They are called *correctness*, *non-completeness*, and *uniformity*. The used terminology is adapted from [30].

- **Correctness:** A so-called realization is a set of functions f_i which together compute the output(s) of f. f is a vector function which is defined as $(z^1, ..., z^q) = f(x^1, ..., x^q)$. This gives the following property:

  Property 1: Let $(z^1, ..., z^q) = f(x^1, ..., x^q)$ be a vector function. Then the set of functions $f_i(\bar{x}^1, ..., \bar{x}^p)$ is a realization of f if and only if

  $$(Z_1, ..., Z_q) = f(X^1, ..., X^p) = \sum_{i=1}^{s} f_i(\bar{X}^1, ..., \bar{X}^p)$$

  for all vectors of input shares $(\bar{X}^1, ..., \bar{X}^p)$ satisfying $\sum_{i=1}^{s} X_i^j = X^j$ with $1 \leq j \leq p$.

  Therefore, the sum of the output of the new function(s) must be equal to the output of the unaltered function for each possible input.

- **Non-completeness:** Given the reduced vector $(x_i^j, ..., x_{i-1}^j, x_{i+1}^j, ..., x_s^j)$ by $\bar{x}_i^j$.

  Property 2: Every function must be independent of at least one share of each component.

  This property is very important for hardware systems. If functions fulfill the *non-completeness* property, glitches do not influence the SS scheme any longer. To fulfill this function a hardware system must deal with at least three shares.

Table 4.2.: Number of influenced gates if a glitch occurs in x_m.

y	m_y	y_m	XOR	AND
0	0	0	0	0
0	1	1	2	2
1	1	0	2	1
1	0	1	1	1

Table 4.3.: Number of influenced gates if a glitch occurs in m_x.

y	m_y	y_m	XOR	AND
0	0	0	0	0
0	1	1	4	2
1	1	0	4	1
1	0	1	3	1

- **Uniformity:** *Property 3:* A realization of $(z^1, ..., z^q) = f(x^1, ..., x^p)$ is uniform, if the distribution of the output shares satisfies

$$Pr(\bar{z}^1 = \bar{Z}^1, ..., \bar{z}^q = \bar{Z}^p) = Q^{1-s} Pr(z^1 = \sum_{i=1}^{s} Z_i^1, ..., z^q = \sum_{i=1}^{s} Z_i^q).$$

  This property implies that if the input is uniformly distributed the output has to be so, too. Otherwise, the result of the function can leak information of the real intermediate value.

4.6.5. Secret Sharing on the Example of Keccak

This section gives three examples making the non-linear function χ used by Keccak CPA secure. All examples are taken from presented literature and use 3 or 4 shares.

Three Share Method

The first approach to make the Keccak permutation immune against CPA was made by Bertoni et al. [5]. For hardware implementations they found a suitable solution shown in Equation (4.3).

$$\begin{aligned} a_i &= b_i \oplus (b_{i+1} \oplus 1) b_{i+1} \oplus b_{i+1} c_{i+2} \oplus c_{i+1} b_{i+1} \\ b_i &= c_i \oplus (c_{i+1} \oplus 1) c_{i+1} \oplus c_{i+1} a_{i+2} \oplus a_{i+1} c_{i+1} \\ c_i &= a_i \oplus (a_{i+1} \oplus 1) a_{i+1} \oplus a_{i+1} b_{i+2} \oplus b_{i+1} a_{i+1} \end{aligned} \tag{4.3}$$

a, b, and c are the different shares, i is taken modulo 5 and runs in x direction. A very nice aspect of these three equations is the fact that the logical operations are the same. Therefore, an iterative usage is possible and hardware area can be reduced by doing so. Nevertheless, not all of the previously mentioned properties are fulfilled. The *uniformity* is violated [11] and it should be possible to mount a CPA successfully.

Three Share Star Method

Due to the fact that Equation (4.3) violates only the *uniformity*, an updated version was presented by Bilgin et al. [11]. In this new function, they add (XOR) a random value to the result of χ. Because of the fact that random values are uniformly distributed, the result of the addition is it, too. The high amount of needed random values is a big drawback of these new functions. If Keccak-f[1600], is used it needs $1600 * 2 = 3200$ random values each round, 1600 for share a and b. The update values for share c is the XOR combination of the values for a and b. Hence, additional storage elements have to be added. The authors also showed that this number can be reduces to 4 bits per round.

Four Share Method

To get rid of the random values, a new version of χ was introduced in the same paper [11]. Now four shares have to be used to achieve all three properties (*correctness*, *non-completeness*, and *uniformity*) and the functions are different for all shares and for some bits. For $i = 0, 1, 2, 4$ the functions follow Equation (4.4).

$$\begin{aligned}
a_i &= b_i \oplus b_{i+1} \oplus ((b_{i+1} \oplus c_{i+1} \oplus d_{i+1}(b_{i+1} \oplus c_{i+1} \oplus d_{i+2})) \\
b_i &= c_i \oplus c_{i+2} \oplus (a_{i+1}(c_{i+1} \oplus d_{i+1}) \oplus a_{i+1}(c_{i+1} \oplus d_{i+1}) \oplus a_{i+1}a_{i+2}) \\
c_i &= d_i \oplus d_{i+2} \oplus (a_{i+1}b_{i+2} \oplus a_{i+2}b_{i+1}) \\
d_i &= a_1 \oplus a_{i+2}
\end{aligned} \tag{4.4}$$

If it comes to index 3 they follow Equation (4.5).

$$\begin{aligned}
a_3 &= b_3 \oplus b_0 \oplus c_0 \oplus d_0 \oplus ((b_4 \oplus c_4 \oplus d_4(b_0 \oplus c_0 \oplus d_0)) \\
b_3 &= c_3 \oplus a_0 \oplus (a_4(c_0 \oplus d_0) \oplus a_0(c_4 \oplus d_4) \oplus a_0a_4) \\
c_3 &= d_3 \oplus (a_4b_0 \oplus a_0b_4) \\
d_3 &= a_3
\end{aligned} \tag{4.5}$$

Again a, b, c, and d are the different shares and i is the identifier for the row bit. Because of the different functions, an iterative method is not possible any longer. But this SS does not need random values. On the other hand, a new share is involved. Hence, again more hardware resources are needed.

Chapter 5

Hardware Architecture of Zorro

Recently, the number of used wireless devices increased steadily. In addition, more and more sensitive data get processed by these devices. In order to secure the processing, cryptographic algorithms can be used. Dedicated hardware implementations can speed up the cryptographic computations. But using dedicated hardware implementations of cryptographic algorithms on wireless devices can lead to problems. First, the additional power consumption caused by the extra hardware circuits can decrease the lifetime of wireless devices which are battery powered. In order to decrease this additional power consumption, low-area designs can be used. A second problem that appears is the availability of wireless devices. Wireless devices can be accessed by a public domain. If attacks are used which focus on implementation facts, this opportunity can be used to attack the cryptographic computation performed on wireless devices. In order to prevent implementation attacks, implementations of cryptographic algorithms must be secured with countermeasure. Altogether, low-area implementations secured against implementation attacks are required for wireless devices.

In this chapter, we want to introduce our low-area Authenticated Encryption (AE) system based on Keccak which is secured against implementation attacks. The system is secured with state-of-the-art countermeasures. We decided to implement both hiding and masking countermeasures in order to increase the resistance against implementation attacks. In particular, we implemented a randomization mechanism and used Secret Sharing (SS) techniques for masking intermediate values. Due to the fact that recently several SS approaches were published (see [11] and [5]), the taped-out Application-Specific Integrated Circuit (ASIC), called Zorro consists of three different independent instances. Each instance only differs with regard to its SS implementation.

The chapter is structured as follows. In Section 5.1, the global requirements on the ASIC are given. Section 5.2 gives a detailed view of the architecture of Zorro. Synthesis results of the implementation are stated in Section 5.4. In Section 5.5, we present back-end results of Zorro. Section 5.6 compares our results with related work.

5.1. Global Requirements and Design Decisions

Zorro was designed to fulfill two main requirements. First, the architecture should result in a low-area design of an AE system based on Keccak-f[1600]. Keccak is a sponge-based algorithm. The sponge-based approach is a new approach for cryptographic primitives to process data. Keccak won the Secure Hash Algorithm (SHA)-3 competition in 2012. Thus, sponge-based algorithms became more and more important recently. The second requirement was to evaluate state-of-the-art countermeasures against power analysis. With this evaluation opportunity we want to answer open research questions such as: How effective are hiding and masking countermeasures on Keccak? What are the costs for SS on low-resource hardware designs?

In this section, we explain the design decisions we made and how we achieve them. The first part of this section is about global hardware decisions we made to keep the area of the architecture low. Furthermore, we present the different modes Zorro can run to evaluate countermeasures against a power analysis.

5.1.1. A Low-Area Hardware Design

In the following, we list several design decision we made to keep the area requirements as low as possible.

A basic requirement of an Integrated Circuit (IC) is to communicate with its environment. In the case of Zorro, we decided to implement a four way handshake protocol (see Section 2.7). In general, the area of the communication interface grows with the size of the data width. We therefore decided to use a data width of 8 bits, which keeps the area requirement of the communication interface low.

A common approach to control a datapath is to use instructions. They can be executed in an arbitrary form. This leads to an extremely flexible system and several sequences are possible. On the other hand, it costs additional area. A Finite State Machine (FSM) is a more hard wired controlling unit. It can only be used for predefined sequences. This reduces the area of the controlling unit, but also decreases the flexibility. In our case, the flexibility is not of relevance. Therefore, we decided to control our datapath with an FSM.

In the case of Keccak, two different constants are needed for the permutation. The first constant is needed for the ρ function and contains the shift values. The second constant belongs to the ι function. The values are the different 64-bit constants of each round. They can be implemented using Linear Feedback Shift Registers (LFSRs). LFSRs, however, require many storage elements which makes them not well suitable for low-area designs. To reduce the size of these units, we therefore decided to implement a Look-up Table (LUT) where logical units with fixed inputs can provide the needed values.

All SS schemes concerning Keccak mentioned so far, work on the algorithm level (see Section 4.6.5). In doing so, the state grows with each used share. Very often the storage elements that are needed for each share have a main influence on the whole area occupation of a cryptographic unit. Random-Access Memory (RAM) macros can decrease the area of such elements in comparison to Flip-Flops (FFs). The additional cycles needed to load the state are a disadvantage of a RAM based approach, but the execution time is of secondary concern for our targeted application. For this reason, ZORRO stores the state in RAM macros. Moreover, only RAMs with a data width of 8 bits are used.

Each en-/decryption needs new random values if it uses a SS scheme. KECCAK initializes its state with zeros at the beginning. To ensure this for a SS scheme, the sum of all shares must be zero. In the case of KECCAK-f[1600] and 3 shares, 3200 random values are needed. We decided to load these random values form a trusted external source (generation of random numbers are out-of-scope of this thesis.)

5.1.2. Countermeasure against Power Analysis

ZORRO offers two different countermeasures: hiding and masking. To evaluate them separately, the implementation should be able to activate and deactivate them separately. Consequently, the IC can run in different modes:

Normal Mode (NM): In this mode, ZORRO en-/decrypts without countermeasures enabled.

Hiding Mode (HM): ZORRO uses hiding as protection against power analysis. We randomized each round of KECCAK by inserting up to 15 dummy rounds. The actual number can be defined by the user. In addition, the implementation shuffles the computation.

Masked Mode (MM): Running in this mode, the en-/decryption is performed with a Secret Sharing (SS) masking scheme.

Secure Masked Mode (SMM): If this mode is activated, SS and hiding are combined.

In Chapter 4, three different SS schemes were presented. To give a widespread spectrum of evaluation options, all three schemes are implemented. Each scheme is represented by an instance, where each instance can operate in all modes. Therefore, they are nearly equal except for the masking scheme. The names of the different instances are:

3-Share instance: This instance uses Equation (4.3) for the non-linear function.

3-Share-Star instance: Using this instance, the update scheme as described in Section 4.6.5 is used for χ.

4-Share instance: Here, χ is executed as outlined in Equation (4.5).

5.2. The Hardware Architecture: A Top-Down Approach

In this section, we explain the hardware architecture of ZORRO in detail. The different masking scheme instances on the IC are nearly the same and only differ because of the SS implementation. The explanation is therefore focusing in the 3-Share instances. We sum up all differences between the three instances later in this chapter.

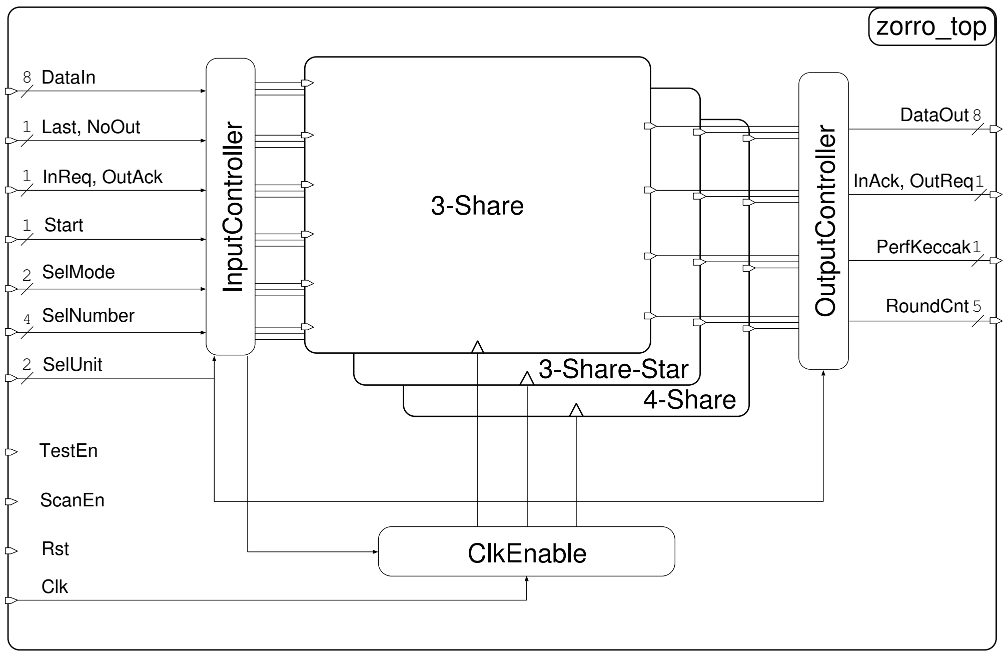


Figure 5.1.: Top module of the ASIC called ZORRO.

5.2.1. Top Level Design

The top level design of ZORRO can be seen in Figure 5.1. On the left side of the figure, the input connections of the ASIC are shown. They are used to activate the desired instance and mode, communicate with the user, and provide a clock signal. In addition, some test and control signals are available. The first block, the inputs come to, is the *InputController*. This unit is responsible for providing the inputs to the enabled instance. The same duty has the *OutputController*, but it is responsible to provide the output signals to the user. The *ClkEnable* entity consists of clock gating units. They are present to activate only the wanted instance. Therefore, only one instance is clocked at the same time. This gives the possibility to evaluate the different instances independent from each other with regards to is power consumption. The output signals are given on the right side and consist of the handshake, the output, and some debug signals.

5.2.2. The Entities of an Instance

The entities of an instance are shown in Figure 5.2. They consist of a RAM macro, the FSM, the datapath, two LUTs for the constants, and an LFSR.

The main part of the instance is the FSM and the datapath. They are responsible for the communication and the en-/decryption. The en-/decryption is done using the KECCAK-f[1600] permutation and is based on the *duplex construction*. The block size is defined with 1088 bits. This leads to a security level of 256 bits.

The RAM is used to store the whole state. The state has a size of 600 bytes in the case

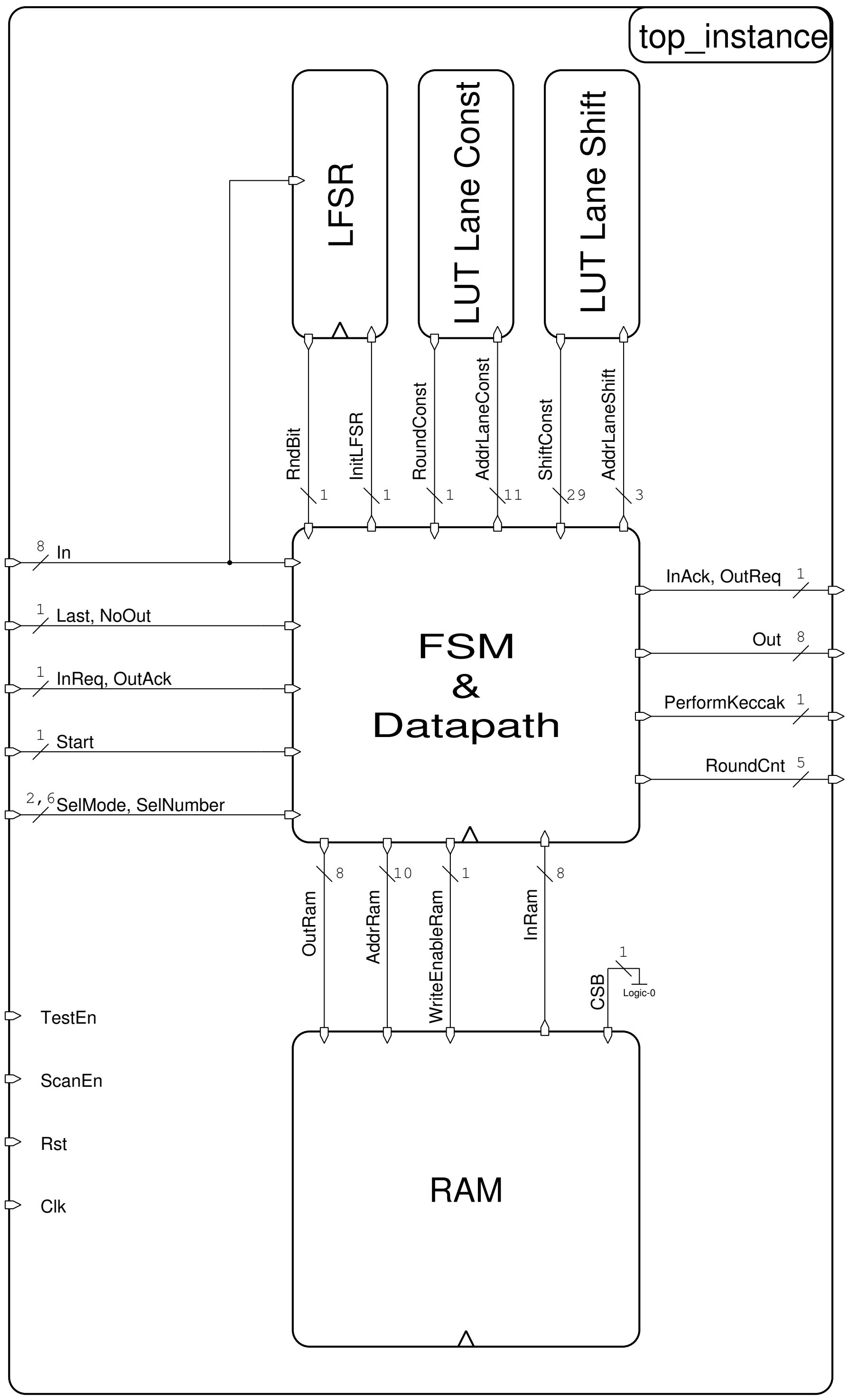


Figure 5.2.: Architecture of the 3-Share instance.

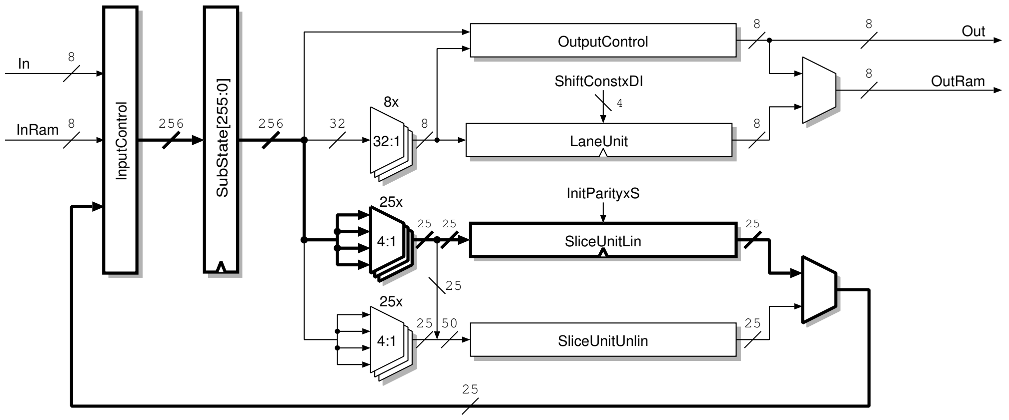


Figure 5.3.: The datapath of the 3-Share instance. The bold line shows the critical path of the instance.

of a 3 share SS scheme. ZORRO provides 8 additional bytes. They are used uninitialized as inputs for the dummy operations.

The LFSR is used to produce random values for the HM. It follows Equation (5.1).

$$x^{32} + x^7 + x^3 + x^2 + 1 \bmod 2 \tag{5.1}$$

It is a maximal length LFSR. The initialization of the state is done with random values of the user. In HM, ZORRO can perform 15 dummy operations. Each dummy operation represents a full KECCAK round. A random bit is used to distinguish whether to insert a dummy operation or not. So up to $15 + 24 = 39$ random bits are needed. The offset for the shuffling is also provided by this LFSR. On that account, it needs 3 additional bits per permutation. Altogether 42 bits are needed per absorbed block. The period of the LFSR is $2^{32} - 1$ cycles. Therefore, random values are available to work off about 99000000 blocks each 1088 bits. This leads to about 100 GB which can be en-/decrypted without a replication of the random values.

5.2.3. The Datapath and FSM

In this section, we describe the implemented datapath in detail. We also describe the implemented FSM and propose a modified scheduling of KECCAK.

The Intermediate Register *SubState* to buffer Lanes and Slices

The different functions used in KECCAK can be split in two groups. The first group (χ, π, θ, and ι) works with whole slices or parts of them (χ combines bits of a row in a non-linear way, π shifts each slice bit by a predefined value, θ sums up 11 bits of two slices, and ι XORs a constant bit to a slice bit, see Section 3.2). In contrast, the second group (ρ) works with lanes (it shifts each lane by a predefined value, see Section 3.2). If the architecture is able to load these parts of the state in a reasonable time, the computation time can be decreased. For the storing of the different parts, a register must be added. The minimum request to it is that it can store at least 1 lane for ρ and 1 slice for the other functions. Thus, at least 64 bits are required. But such a small size does not allow to load slices and lanes at full speed. Consider that the state is linear in the RAM. Therefore, each word of the RAM consists of information of 1 lane and 8 slices. To load slices at full speed, we need a register for 8 slices and it has to have a size of 200 bits. But this is only valid for the NM. In MM, we use Equation (4.3) for χ. This leads to a register of $3 * 8 * 25 = 600$ bits. 600 bits are 33 % of a single state. The area advantage we got because of the used RAM is decreased in a high dimension by the relatively big register with this approach.

Another approach is to store the state soaring from slice to slice. The result is that each word of the RAM stores information of 1 slice and 8 lanes. Now loading lanes at full speed requires a register of $64 * 8 = 512$ bits. This is an improvement, but it is still not satisfying.

The next approach is to interleave bits of lanes and slices. The state is not stored in a linear manner any longer. Consider that each word of the RAM contains 4 bits of 2 slices which is equal to 2 bits of 4 lanes. Loading slices at full speed requires a register of $2 * 25 = 50$ bits, for lanes we need $4 * 64 = 256$ bits. We were able to reduce the amount of storage elements from 512 to 256, but we are still able to load lanes and slices at full speed. A small problem appears because of the state size. The state consists of 25 lanes. Therefore, one lane cannot be stored in this interleaved form. We decided to store the first lane in a linear way because this lane is not influenced by ρ and can therefore be skipped for this function. The drawback of this solution is that slices must be loaded in 8 slice chunks, otherwise cycles are wasted. In NM, we have no problem because we can store 8 slices at the same time in the 256-bit register. In MM, this leads to a problem because $3 * 8 * 25 = 600$. We decided to keep the register size of 256 and store only 2 slices of each share at the same time in the register in MM. The result is that we cannot load slices at full speed in this mode. But this influences only slice bits of the first lane. In NM, we can load 8 slices in 25 cycles. For two slices in MM, we need 7 cycles. This also concerns the SMM. Altogether, we do not waste many cycles in comparison to the saved area. The same register is used to buffer data from the user. The basic principle of interleaved storage was published in [10]. Pessl and Hutter [32] used this approach combined with RAM macros in.

With this we are able to load slices and lanes in a reasonable time and can also buffer data from the user. Figure 5.3 shows an overview of the datapath used by the 3-Share instance. Here the register to buffer data from the user or the RAM is called *SubState*. The *InputControl* entity in this figure is responsible to load the different state pieces and the input data into the register. The loading of the state pieces is done via a shift into the *SubState* register. The *OutputConrol* entity provides initialization data into the RAM or results to the user.

Slice Based Functions

Due to the interleaved storage structure and the *SubState* register, we are able to load slices and lanes in a reasonable time. The permutation of KECCAK uses 5 different functions. Four of them (θ, χ, ι, and π) take whole slices or parts of them as input. If ZORRO is able to compute the result for these functions of a whole slice at once, the permutation time can be reduced. Therefore, the first goal was to alter them to fulfill this requirement. In the following, we describe the implementation of the permutation in a more detailed way.

Theta Permutation θ: Each execution of θ manipulates 1 bit of the state. As input it takes 11 bits of 2 different slices. With 25 instances in parallel the result of a whole slice can be calculated. Because θ uses information of two slices at the same time, we installed a register to store the information of one slice. So the implementation only needs one slice at the same cycle of the *SubState* register. The needed initialization of the register is a drawback of this solution. But in total it leads to an advantage and the waisted cycles can be accepted.

Chi Permutation χ: χ works with a row. If 5 instances of this function are taken, the result of one slice can be calculated in one cycle. Therefore, the datpath of ZORRO uses 5 instances of χ in parallel.

Pi Permutation π: π shifts all bits of one slice by a predefined value. So this function can already handle a whole slice and must not be altered.

Iota Permutation ι: ι is a XOR operation of a 64-bit constant with a lane. But the constant can be split in 1-bit chunks and can then be applied to each slice. To achieve this, our LUT, called *LUT Lane Const* (see Figure 5.2), provides 1 bit per cycle and address.

Altogether, now the 4 functions (θ, π, ι, and χ) work with whole slices. They are concatenated and can calculate the result of one slice in a single cycle. In Figure 5.3, the *SliceUnitLin* entity computes the result of these functions. It takes a whole slice as input. If the instance runs in MM, χ must follow Equation (4.3). To achieve this, an additional entity is added. In Figure 5.3, the *SliceUnitUnlin* is this entity. It takes two shares as input and outputs the result of one share in a single cycle. Because the function for χ is symmetric, it can be used iteratively for each share. The multiplexers in front of these two entities are responsible to choose different slices which are stored in the *SubState* register.

Lane Based Function

We already have taken care of 4 out of 5 functions. The remaining one, ρ, shifts each lane by a predefined value. These values differ from lane to lane. The architecture can load 4 lanes at the same time into the *SubState* register. A lane consists of 64 bits and ZORRO uses RAM macros with a data width of 8 bits. If we are able to compute the

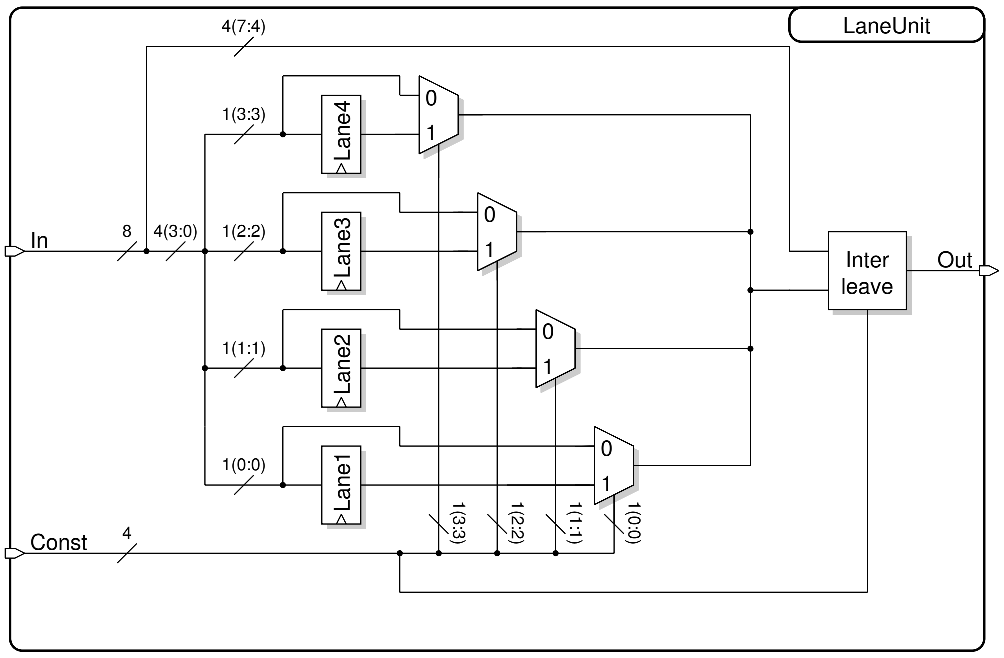


Figure 5.4.: Block diagram of the optimized ρ function of Zorro.

result of ρ byte per byte, the calculation step can be combined with the saving into the RAM. This can be achieved in the following way:

The architecture can shift in 2-bit steps easily. It stores 2 bits of a lane in 1 byte of the RAM. Loading from address n and storing the same byte back to address $n+1$ results in a shift of 2 bits. Therefore, shift values of a multiple of 2 can be compensated with simple address variation.

The next step is to compensate the differences between the shift values of the 4 lanes. We call 2 bits of a lane which are stored in the same byte a bit couple $bc_{z,y}$, where z is the index of the bit couple and y the index of the lane. The different values for z and y run from 0 to 31 and from 1 to 24, respectively. The differences between shift values of several lanes can be compensated by choosing contrasting bit couples. Consider, we have loaded the first byte of the 4 lanes from address x. 4 lanes have $64 * 4 = 256$ bits and therefore the last byte was loaded from address $x + (256/8 - 1) = x + 31$. The bytes get stored soaring in our *SubState* register. So SubState(7:0) corresponds to the value of address x and SubState(255:248) to the value of address $x + 31$. Assume that the i^{th} byte ($0 \leq i \leq 31$) of the register should be shifted. This byte was loaded from address $x + i$. We can choose different bit couples for each lane. Which couples of the lanes are taken can be calculated as follows. For each of the 4 lanes the maximum shift value minus its own shift value is taken. The resulting value divided by 2 is taken as bit couple difference (notice that the bit couple difference is for the 4 lanes diverse). The index of the chosen bit couples is calculated with i plus the couple difference. If we store the chosen bit couples back to address $((i + max(shift\_value)/2) \bmod 32) + x$, where

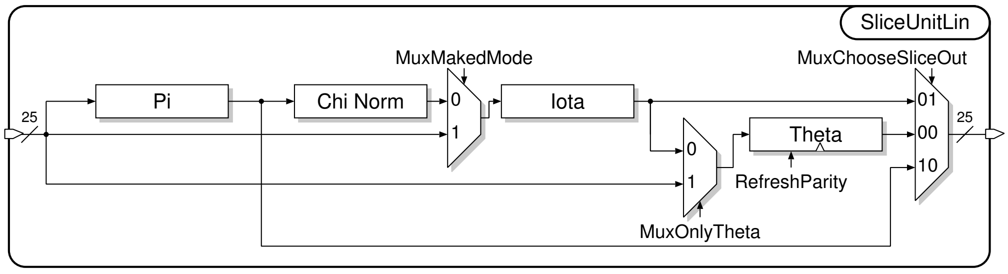


Figure 5.5.: Slice function structure of ZORRO without χ for the MM.

$max(shift\_value)$ is the maximum of the shift values for the loaded lanes, a shift up to 62 bits is possible and the shift value differences are already compensated. In the case of the first byte of lane 1 to 4, the shift values are 1, 62, 28, and 27. The $max(shift\_value)$ is 62 and i is 0. We can choose $bc_{31,1}$, $bc_{0,2}$, $bc_{17,3}$, and $bc_{18,4}$. A storing back to address $x + 31$ is required. By performing this, we shift the first byte of the loaded lanes by the right constants. In practice, i must run from 0 to 31 to perform the shift for all loaded bytes.

Up to now, we have compensated the offset of the lanes and a shift in 2-bit steps of maximum 62 bits is possible. A problem appears if a shift value is not a multiple of 2. The solution is a 1-bit register for each lane. If $shift\_value$ mod $2 \neq 0$, the high bit is saved into this register. For the next bit couple of this lane, the stored bit can be the low bit of it. The original low bit serves now as high bit. This can be made for the bit couples of all lanes. The registers have to be initialized which is a drawback. But altogether these alterations of ρ lead to an extremely small entity. It only needs 4 registers and 4 multiplexers to decide if this one bit shift is performed or not.

The values (couple differences, maximal shift value divided by 2, and shift values modulo 2) it needs can be precalculated and stored in a LUT. In our case, we stored them in a LUT called *LUT Lane Shift* which can be seen in Figure 5.2. In Figure 5.3, the entity *LaneUnit* is responsible for ρ. The multiplexers in front of it are responsible to choose the different bit couples. A more detailed view of the *LaneUnit* can be seen in Figure 5.4. The *InterLeave* block is responsible to exchange the high with the low bit if the one bit shift is performed. This is done for each bit couple.

The Round Schedule of ZORRO

The original schedule of KECCAK is θ, ρ, π, χ, and ι. The first operation operates with slices, the next with lanes, and afterwards, again functions which work with slices are present. Switching from slices to lanes or the other way round results in storing back the whole information into the RAM. In the original scheduling, we have to switch 3 times per round. A reorganisation to π, χ, ι, θ, and ρ reduces, the loading and saving sequences of the whole state from 3 to 2. Due to the fact that the result has to be the same after the reschedule, an additional round has to be added. As a result, our implementation needs 25 rounds. A detailed view of the different operations and rounds can be seen in

Equation (5.2). This reschedule is the same as made by Pessl and Hutter [32]. Another reschedule concerning KECCAK was presented by Jungk and Apfelbeck [16].

$$\begin{aligned} R_1 &= \theta \times \rho \\ R_{2\ldots24} &= \pi \times \chi \times \iota \times \theta \times \rho \\ R_{25} &= \pi \times \chi \times \iota \end{aligned} \tag{5.2}$$

Figure 5.5 shows the *SliceUnitLin* in detail. The different multiplexers are present to bypass parts of the concatenated functions. It must be able to compute only θ (round 1), only $\pi \times \chi \times \iota$ (round 25), and skip χ (MM).

In MM, the architecture must compute the permutation for each share and another χ function is used. For this function information of 3 shares are needed. When χ gets computed, all shares have to have passed π. To achieve this within a single cycle, 2 additional θ, ι, and π units are necessary. Another approach is to add a loading and saving sequence. Our architecture uses this approach and adds a loading and saving sequence between the function pairs $\theta \times \pi$ and $\chi \times \iota$. ZORRO loads the shares consecutively for $\theta \times \pi$. After the computation, it stores the information back into the RAM. For $\chi \times \iota$ it loads 2 slices of each share, computes the result, and stores it back. This is done sequentially for each slice pair. This additional loading and saving step increases the computation time of the slice operations. Altogether these alterations lead to a computation time increase greater than linear with the share quantity in MM compared with the NM. But the area occupation gets reduced with this approach. The ρ function can be executed iteratively on each share.

5.3. Comparison between the Three Instances

In the last section, we describe the architecture of one instance of ZORRO. All facts belong to the 3-Share implementation. In this section, we compare the different instances.

3-Share vs 3-Share-Star

Between these two implementations only the update step after χ is different. For the 3-Share instance, Equation (4.3) is used for the non-linear function. But this function does not provide the third property, *uniformity*, we introduced in Section 4.6.4. Bilgin et al. [11] an update scheme for the function. 10 random bits per slice are XORed to the result of the function. Working with 3 shares results in 20 random bits. In MM, two slices of each share are stored in the *SubState* register at the same time. The computation of χ takes one cycle for each slice. Consequently, 40 bits are needed in two cycles which have to be random. They are gained with the help of the LFSR. This gives the possibility to get one bit per cycle. Thus, another register has to be added to store these 40 values. The saving and loading of the slices take more than 40 cycles. Therefore, it only costs us the register and we do not loose any cycles by creating the random values. For each block $20 \times 64 \times 24 = 30720$ additional random values are needed. This reduces the en-/decryption size to near 140 MB without replication of the random values.

3-Share vs 4-Share

These two instances differ because of the share quantity and the χ function itself (see Equation (4.5) and Equation (4.4)). Due to the fact that the share quantity is increased by one, the execution time and the area increase. A small amount of the additional area can be traced back because the functions for the different shares are not symmetric any longer. The biggest part of the increase comes from the bigger RAM. Additionally, because of the different structure of χ, the intermediate register has to be increased to 300 bits. The communication between the user and the ASIC is the same.

Summarized, this implementation needs more cycles and has a bigger area occupation in comparison to the 3-Share implementation.

5.4. Synthesis Results of Zorro

Zorro was implemented using Very High Speed Integrated Circuit Hardware Description Language (VHDL). For the functional simulation, *Mentor GraphicsModelSim 10.2a* was used. The synthesis was made with *Synopsys Design Compiler 2012.06.* The standard cell library the design was mapped to is based on the 180 nm Complementary Metal Oxide Semiconductor (CMOS) technology processed by United Microelectronics Corporation (UMC). When it comes to a Gate Equivalent (GE) this library takes a 2-input NAND gate as reference. This gate has an area of $9.3744\,\mu\text{m}^2$.

The results we want to present are synthesis results. Figure 5.6 shows an area-time plot of Zorro. A decrease of the period from 20 ns to 7 ns does not increase the area of the chip. Thereafter, the GE count starts to increase. The most efficient results are gained from 3.05 ns. Our system was designed to reduce the area of the implementation. Therefore, efficiency is not that important and we decided to focus on the results gained at 4 ns (250 MHz).

Critical Path

Zorro concatenates all slice functions and stores 8 bits at the same time in the *SubState* register. To choose the right slice, multiplexers are used in front of the *SliceUnitLin* entity. The values are stored back by the *InputControl* unit (see Figure 5.3). Altogether, this leads to big multiplexer networks around the register. The path from the register through the *SliceUnitLin* unit and back to the register defines the critical path and therefore the maximum frequency of the ASIC. Figure 5.3 a shows schematic view of the datapath used by the 3-Share instance. The bold line highlights the critical path of the instance. Additionally, a detailed apportionment is shown in Table 5.1.

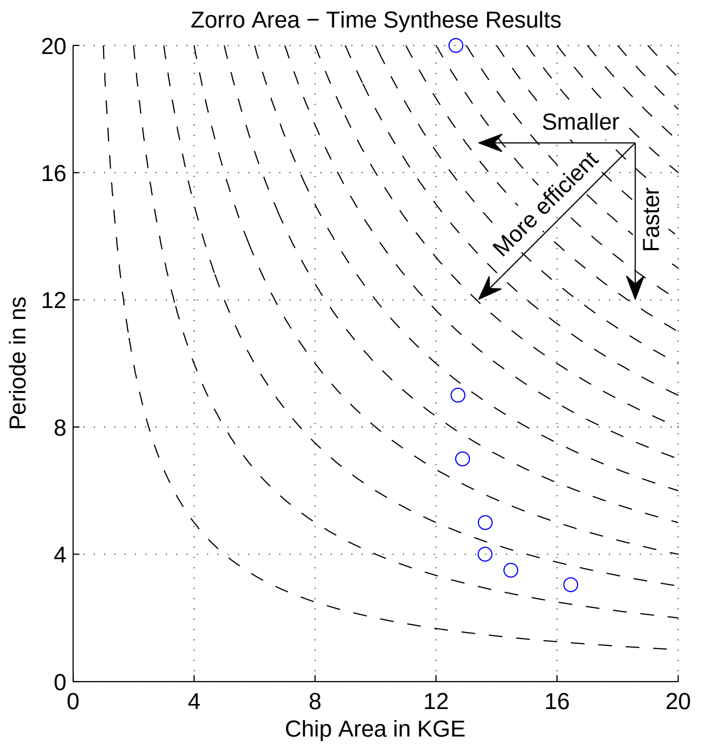


Figure 5.6.: The area-time plot of ZORRO after synthesis.

Area

The main storage element we used in the design was a RAM macro. Because of the big memory amount, a big part of the whole area is occupied by it. In numbers, this belongs to 5000 to 6000 GEs for each instance.

The design decisions we made (disarm the complexity of ρ and iterative processing) gave us a very small area for the original KECCAK functions. Altogether they only need around 650 GEs. The LUTs have a GE count of 250 and are equal for each instance.

An abnormality appears concerning the *SliceUnitUnlin* entity. This unit contains the χ function for the MM. For the 4-Share instance it is nearly three times bigger than for

Table 5.1.: The critical path of ZORRO after synthesis for a maximum frequency of 250 MHz.

Activity	**Time**
SubState to *SliceUnitLin* input	2.21 ns
SliceUnitLin in- to output	1.36 ns
SliceUnitLin output to *SubState*	0.43 ns

the other two. This has the following reasons. First, now four shares are involved and up to three shares are needed at once for the computation. This increases the logical amount for the functions. Additionally, they are not symmetric any longer. Consequently, each share needs its own function which again increases the area amount of this unit. It needs 250 GEs for the two 3 share implementations and 750 GEs for the 4 share implementation.

The LFSR with a state of 32 bits needs 350 GEs. The *SubState* register plus the surrounding logic need 4500 GEs in the case of the 3-Share instance. For the 3-Share-Star instance, the value increases to 5100. This can be traced back to the additional XOR gates and registers needed for the new χ function. For the 4-Share instance, the number is again increased to 6200. The bigger *SubState* register and the increased multiplexer networks around it are the explanations for that.

The FSM takes up 3000 GEs for each implementation. This is a relatively high GE count for an FSM. In our design, each instance can run in different modes. This leads to a huge amount of states which is the explanation for the increased size. If an implementation only supports SS as countermeasure, the size of the control unit is reducible.

Altogether the 3-Share instance needs around 14000 GEs, the 3-Share-Star instance 14600 GEs, and the 4-Share instance 17200 GEs. A detailed apportionment of the whole chip can be found in Table 5.3.

5.4.1. Timing

All instances can compute a round in 912 cycles in NM. This value is comparable to the original KECCAK description. Therefore, a full permutation can be achieved in $24 \times 912 = 21888$ cycles. In MM, the three share implementations need 113184 cycles. This is can increase of more than 5. The increase of the state size from 1600 bits to 4800

Table 5.2.: Cycle count for the different activities ZORRO can perform.

Activity	**3-Share** [Cycles]	**3-Share-Star** [Cycles]	**4-Share** [Cycles]
KECCAK round NM	912	912	912
KECCAK round MM	4716	4716	6235
State initialization NM	201	201	201
State initialization MM	2199	2199	3200
LFSR initialization	17	17	17
Input/Output (I/O) communication NM	2040	2040	2040
Output communication NM	952	952	952
Input communication NM	952	952	952
I/O communication MM	2313	2313	2449
Output communication MM	1224	1224	1360
Input communication MM	1225	1225	1363

bits and the additional loading and saving cycle between $\pi\,\chi$ and $\iota\,\theta$ is the explanation for this higher cycle count. The 4-Share implementation needs 149640 cycles in MM for the permutation. This is an increase of one third in comparison to the three share implementations and can be traced back to the additional share.

For HM only the rounds in one permutation increase. Accordingly, the en-/decryption time increases linear with the dummy operations.

The I/O communication is relatively slow. This can be attributed to the four way handshake protocol. In NM, the communication decreases the throughput by approximately 10 %. Because of the increased computation time in MM, the throughput is only decreased by 4 % in this mode.

A detailed cycle count for the different activities can be found in Table 5.2.

5.5. Back-end Results of ZORRO

The goal was to tape out the resulting design. In comparison to the results after synthesis, we had to add additional structures to improve the testability. For the testing of the datapath and the FSM, we used a full scan chain approach. Thus, all registers were exchanged with scan chain registers. Because we do not use a lot of registers the size was not increased in a high dimension. To test the functionality of the different RAMs, the FSM was adapted to provide new states. These states make it possible to write on each address of the RAMs. Afterwards, all addresses can be read out again. The communication is made over the already available handshake protocol.

The main goal was to keep the implementation as small as possible. Therefore, we reduced the maximum frequency to 200 MHz. This compensates the area increase because of the additional test structures. The area results can be found in Table 5.4. The cycle count for the different operations stayed the same as after synthesis.

The back-end design was made with *Cadence SoC Encounter 9.1.* Additional Layout Versus Schematic (LVS) and Design Rule Check (DRC) tests were made with *Calibre DESGINrev 2010.2.* For the test vector generation *Synopsys TetraMAX ATPG 2012.06* was used. Figure 5.7 shows the regions the different instances occupy. The layout of the finished chip can be seen in Figure 5.8.

5.6. Comparison with Related Work

For the comparison with related work, the area occupation and frequency after back-end design is taken. Comparison results of different technology are sometimes hardly meaningful. Therefore, we only focus on 130 nm and 180 nm technologies because they are more ore less comparable to our technology. SS combined with KECCAK is a relatively new topic. But there exist implementations for all of our instances. A difference between our work and related work is that we can en/-disable our countermeasures. In addition, we also support hiding and we produce the random values for it on the IC. Therefore, for a single KECCAK-f[1600] implementation protected with SS the area occupation should be reducible. However, we want to present our results compared with related work in detail.

3-Share instance: The 3-Share instance was implemented using the SS scheme presented in [5]. Their smallest implementation secured with the SS has an area occupation of 95 kGEs. Our 3-Share implementation needs 14 kGEs. Therefore, our 3-Share instance needs approximately 15 % of the size of the architecture presented in [5].

3-Share-Star instance: The 3-Share-Stare instance was implemented using the 3 share SS scheme presented in [11]. Their 3 share SS implementation needs at least 32.6k GEs. Our implementation has an area occupation of 14.5 kGEs.

4-Share-Star instance: The 4-Share-Stare instance was implemented using the 4 share SS scheme presented in [11]. In addition, they presented synthesis results of their implementation. The smallest implementation needs 32.5 kGEs. Our implementation needs 17.0 kGEs.

Non-protected: Compared to the work of [32], it shows that our Correlation Power Analysis (CPA)-protected version of KECCAK is less than 3 times larger in size. The larger size is mainly duo to the increased state. Our implementation needs a state which is 3 times lager compared to [32].

In all cases we were able to reduce the area occupation by more than 50 %. The throughput was reduced by a factor over 1000 compared with related work. Our implementations are extremely trimmed for a small area occupation. Consequently, our throughput compared with other work is secondary. A detailed comparison can be found in Table 5.5.

Table 5.3.: Area results after synthesis without test structures of ZORRO. The maximum frequency is given by 250 MHz.

Component	**Area** [GE]	**Total Percent** [%]	**Sub Percent** [%]
ZORRO	47000	100	100
3-Share	14000	30.0	30.0
FSM & Datapath	8400	18.0	60.0
FSM	3000	6.5	35.7
SliceUnitLin	550	1.2	6.6
SliceUnitUnlin	250	0.5	3.0
LaneUnit	100	0.2	1.2
Inter. Register & Logic	4500	9.6	53.6
RAM	5000	10.6	35.7
LUT Lane Const	150	0.3	1.0
LUT Lane Shift	100	0.2	0.7
LFSR	350	0.7	2.5
3-Share-Star	14600	31.0	31.0
FSM & Datapath	9000	19.2	61.7
FSM	3000	6.5	33.3
SliceUnitLin	550	1.2	6.1
SliceUnitUnlin	250	0.5	2.8
LaneUnit	100	0.2	1.2
Inter. Register & Logic	5100	10.9	56.7
RAM	5000	10.6	34.3
LUT Lane Const	150	0.3	1.0
LUT Lane Shift	100	0.2	0.7
LFSR	350	0.7	2.4
4-Share	17200	36.6	36.6
FSM & Datapath	10600	22.6	61.6
FSM	3000	6.5	28.3
SliceUnitLin	550	1.2	5.2
SliceUnitUnlin	750	1.6	7.1
LaneUnit	100	0.2	1.0
Inter. Register & Logic	6200	13.2	58.5
RAM	6000	12.8	34.9
LUT Lane Const	150	0.3	0.9
LUT Lane Shift	100	0.2	0.6
LFSR	350	0.7	2.0

Table 5.4.: Results after back-end design with test structures of Zorro. The maximum frequency is given by 200 MHz.

Component	**Area** [GE]
Zorro	46000
3-Share	14000
3-Share-Star	14500
4-Share	17000

Table 5.5.: Results for different SS (un)protected Keccak-f[1600] implementations.

Source	**Tech.** [nm]	**Area** [kGE]	**Frequency** [MHz]	**Time** [Cycles]	**Throughput** [Gbit/s]
Bertoni et al. [5][1]	130	127.0	200	-	8.5
Bertoni et al. [5][1]	130	95.0	200	-	2.8
		This work			
Zorro 3-Share	180	14.0	200	113184	$1.88\ 10^{-3}$
Bilgin et al. [11][1]	180	145.3	516	25	-
Bilgin et al. [11][1]	130	135.2	746	25	-
Bilgin et al. [11][1]	180	33.1	553	1625	-
Bilgin et al. [11][1]	130	32.6	820	1625	-
		This work			
Zorro 3-Share-Star	180	14.5	200	113184	$1.88\ 10^{-3}$
Bilgin et al. [11][1]	180	174.2	513	24	-
Bilgin et al. [11][1]	130	157.6	735	24	-
Bilgin et al. [11][1]	180	43.1	572	1600	-
Bilgin et al. [11][1]	130	42.4	633	1600	-
		This work			
Zorro 4-Share	180	17.0	200	149640	$1.43\ 10^{-3}$
Pessl and Hutter [32]	130	5.5	-	22570	-
Pessl and Hutter [32]	130	5.8	-	15427	-
		This work			
Zorro NM	180	14.0	200	21888	$9.09\ 10^{-3}$

[1] Block size of 1024 bits;

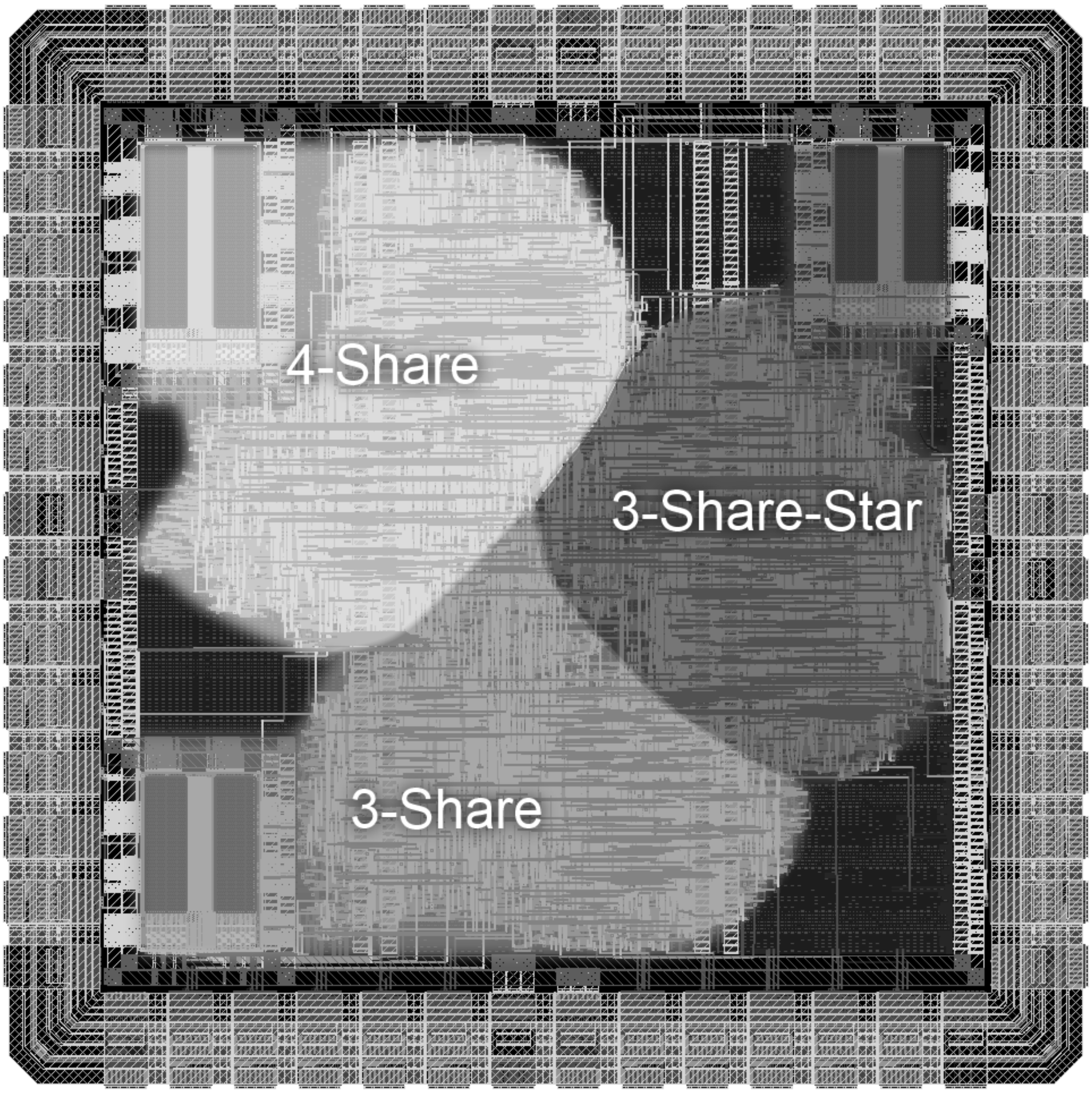


Figure 5.7.: A picture that shows in which regions the different instances of Zorro are.

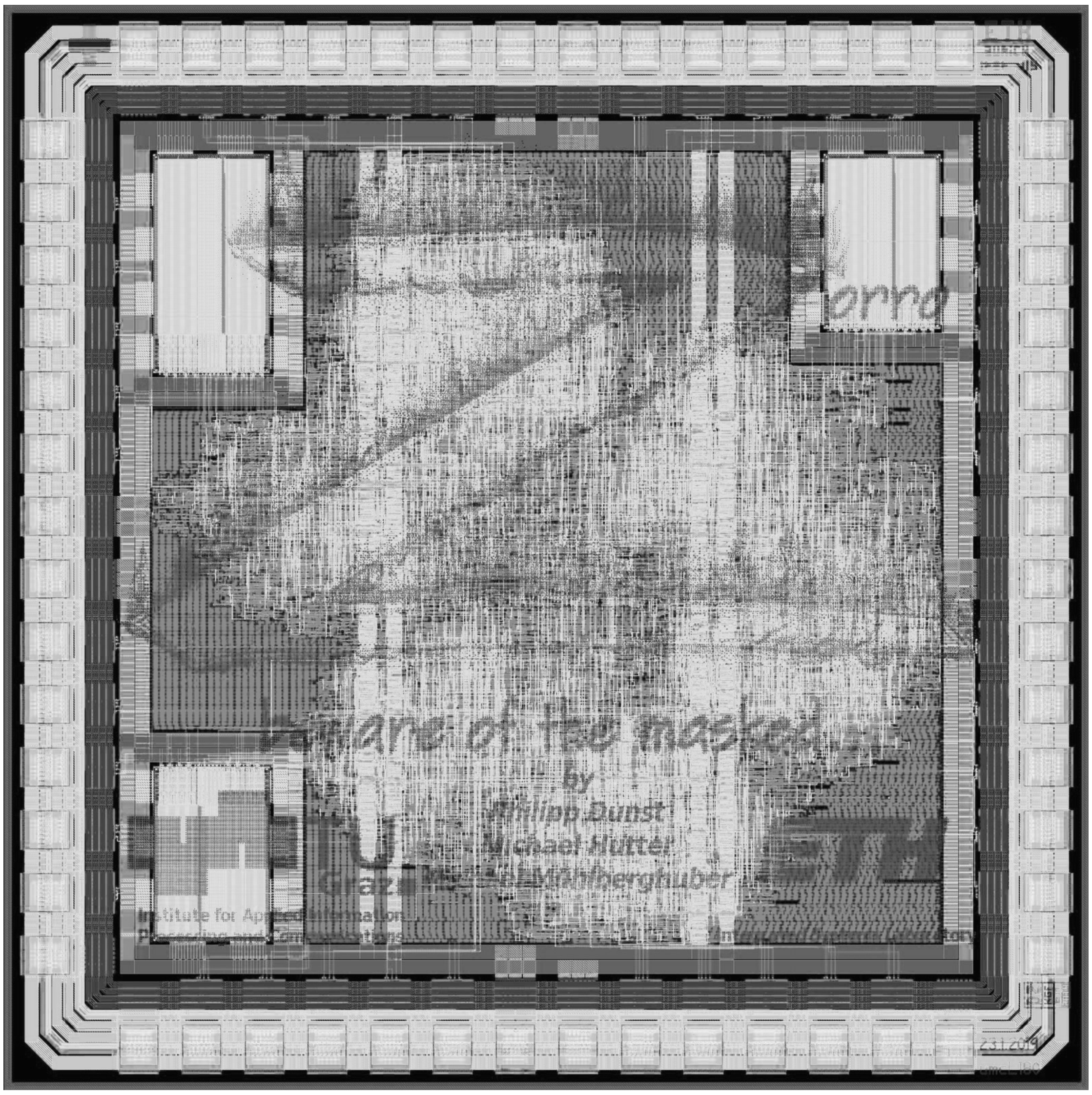


Figure 5.8.: A picture of the final chip layout of Zorro.

Chapter 6

Power Analysis of ZORRO

Nowadays, simulation tools give the possibility to simulate the power consumption of circuits. In this chapter, we discuss the first power simulation of ZORRO. All simulations are made with *Cadence SoC Encounter 9.1* using a Value Change Dump (VCD) file generated with *Mentor GraphicsModelSim 10.2a*. The frequency is defined by 10 MHz, the sampling step size by 10 ns. The analyses are made with *MATLAB 7.9.0*.

The chapter is structured as follows. In Section 6.1, a power analysis of an encryption operation of ZORRO in Normal Mode (NM) is given. Section 6.2 explains the power consumption of an encryption in Masked Mode (MM). In Section 6.3, different Correlation Power Analyses (CPA) against ZORRO are outlined. We start with an attack against the 3-Share instance running in NM. Afterwards, results of attacks against the Hiding Mode (HM) and the MM are presented.

6.1. Power Consumption of an Encryption in Normal Mode

The first target of the analysis is an encryption operation in Normal Mode (i.e., no countermeasures are enabled) of the 3-Share instance. Figure 6.1 shows the simulated power consumption during the encryption. Two input blocks are processed. Five different regions are very eye-catching (separated with doted lines). The first separation belongs to the first communication with the user. The power consumption is comparatively low because not many hardware components are active. Afterwards, the first permutation starts which takes almost 50 % of the whole encryption. In the middle of the plot, a high power consumption across a few milliseconds can be identified. This is again the communication with the user. In contrast to the Input/Output (I/O) communication at the beginning of the plot, the power consumption is much higher. The reason for this is that ZORRO provides also output data. These output data are available on the output pads, which were deactivated at the beginning of encryption. After that, the second permutation is performed. Finally, the output is read out via I/O pads which again increases the power consumption at the end of encryption. Note that the communication

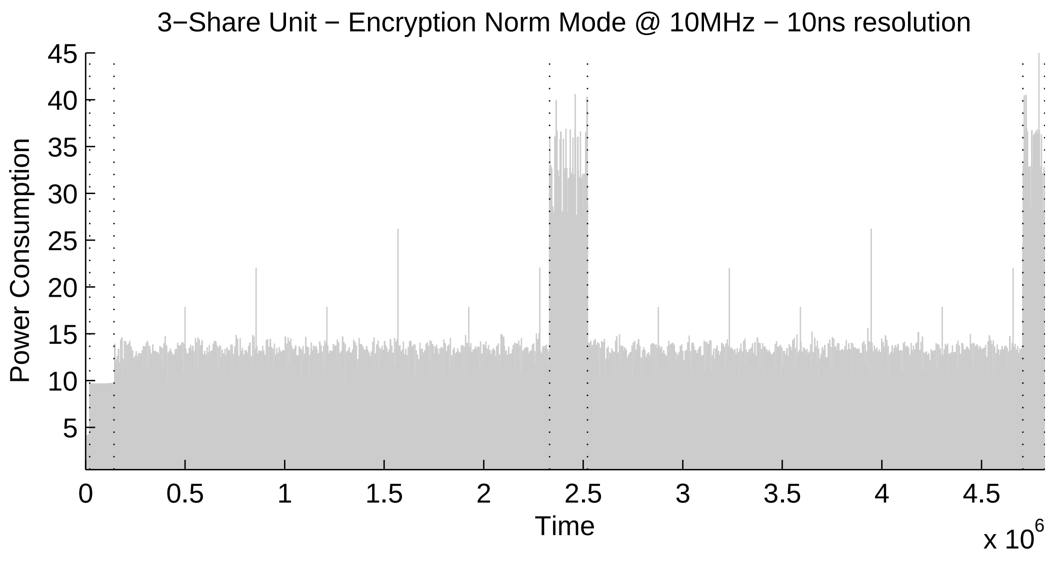


Figure 6.1.: Simulated power consumption during an encryption operation of the 3-Share instance of Zorro running in NM.

is shorter because the user does not send new data (Message Authentication Code (MAC) output) to the Integrated Circuit (IC).

Figure 6.2 shows a zoomed view of the first permutation. The different rounds are separated via vertical doted lines. The only abnormality can be found in round 25, which is the last round. Here the computation time is shorter in comparison to the other rounds. This can be explained by the fact that in this round ρ is skipped and therefore the computation time is shorter.

In Figure 6.3, the estimated power consumption of a single round can be found. The dashed line separates the slice from the lane operations. The doted lines separate the iterations of the functions. The small separation at the beginning shows the initialization from the θ function. For the first iteration, the power consumption looks like a ramp function. This can be explained as follows. When the computation starts, the *SubState* register contains almost only zeros (some bits can differ from zero because of the previous input communication). The loading of the intermediate values is performed as shift operation into the register. Therefore, the more bits are loaded, the more switching operations can happen and the power consumption raises continually. For the following slice operations, the register contains old intermediate values during loading. Consequently , the loading sequence needs approximately the same power consumption for each loading step. By having a closer look at the power consumption trace, it also shows that the first iteration of the lane processing looks different than the other iterations. The reason is contrary to the effect we have identified before. For the first lane operation, the register contains old intermediate values of the last slice operation. The loading of the

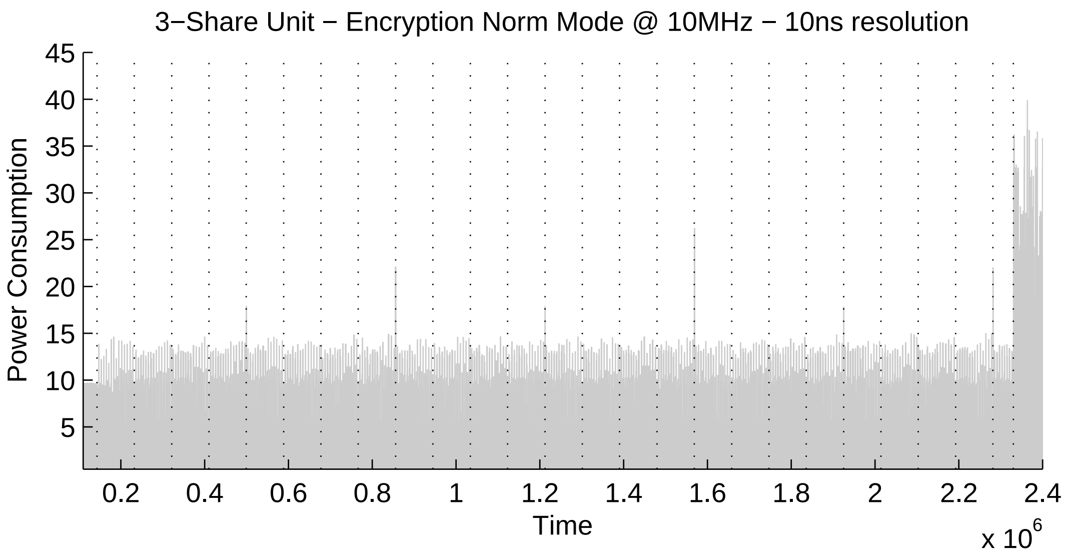


Figure 6.2.: Simulated power consumption during a permutation of the 3-Share instance of Zorro running in NM.

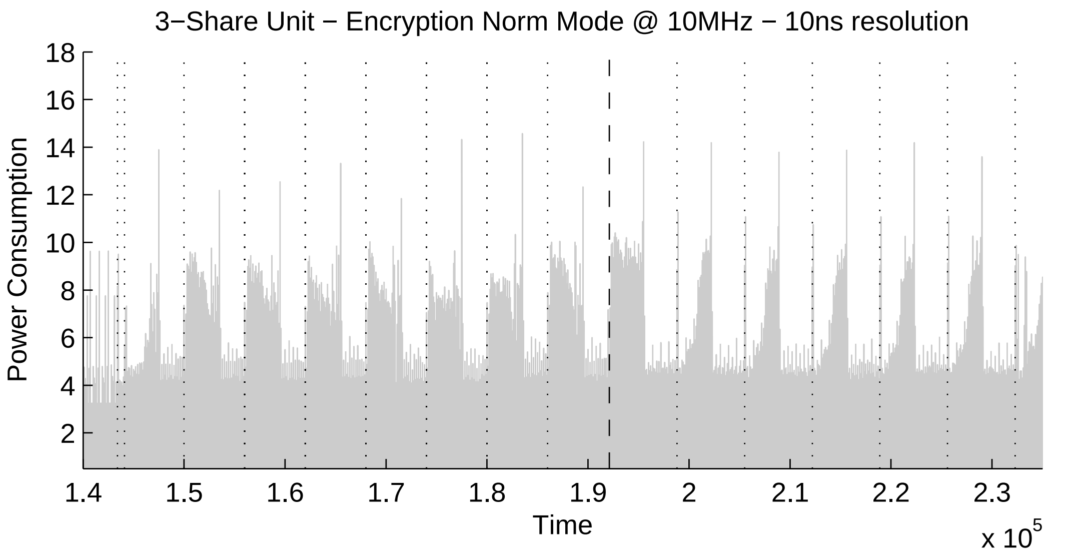


Figure 6.3.: Simulated power consumption during a single round of the 3-Share instance of Zorro running in NM.

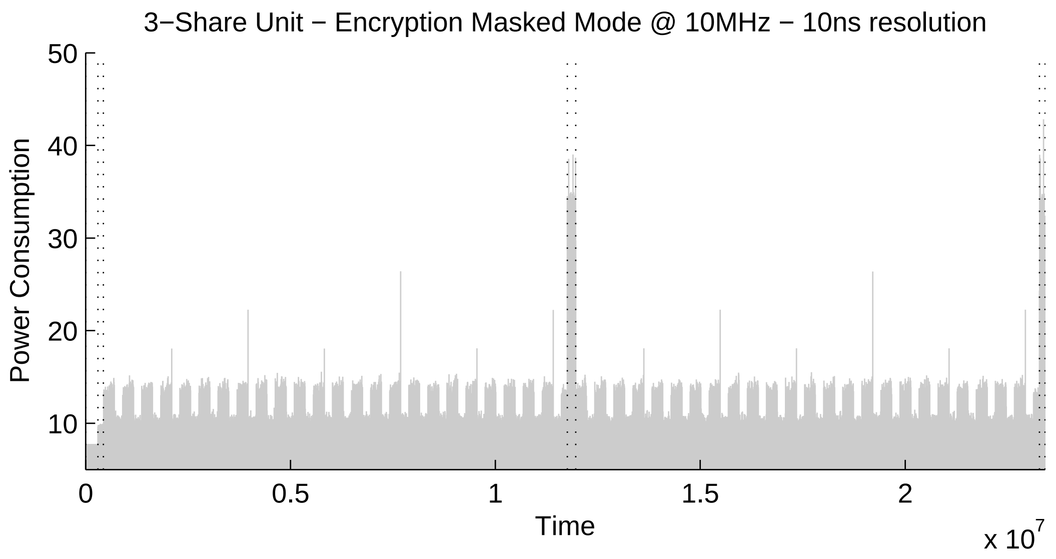


Figure 6.4.: Simulated power consumption during an encryption operation of the 3-Share instance of Zorro running in MM.

lanes is again performed as a shift operation. Sequentially, the power consumption for the first lane operation looks like a rectangle. After each lane operation the register is reset. This is done to compute the result of ρ for each share independently from each other in MM. Because we use the same architecture in all modes, this reset happens also in NM. Therefore, the register contains only zeros at the beginning of the loading sequences. This explains the ramp function shape of the power consumption of the other lane operations. In addition, the high peak at the end of a lane operation can explained by the reset of the register.

6.2. Power Consumption of an Encryption in Masked Mode

The same simulation was done for the Masked Mode (MM). Figure 6.4 shows the result where we can identify five district power-consumption parts. Additionally, we can identify the random state initialization right at the beginning of the plot. The remaining power consumption looks similar to the power consumption during NM.

Figure 6.5 shows the separated rounds of one permutation. Now the first and the second rounds are shorter. This belongs to the fact that the slice operations are split in blocks. The first block computes θ and ι, the second one χ and π. The first round consists only of θ and ρ. So we can skip the second block which leads to a shorter computation time of round one. The last round is again shorter because ρ is skipped. The peak that sometimes appears after a round can be again attributed to the reset of the *SubState* register.

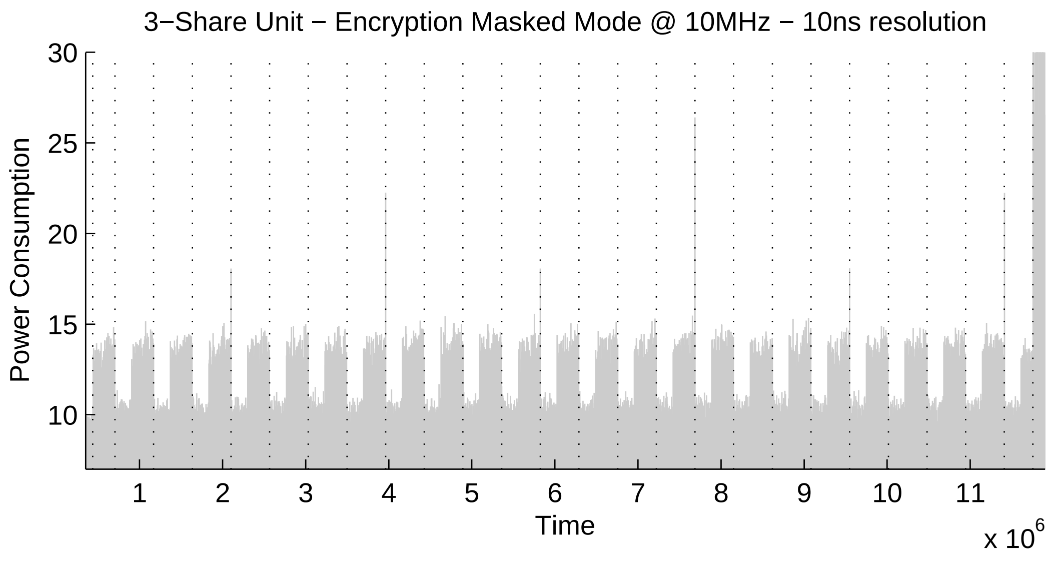


Figure 6.5.: Simulated power consumption during a permutation of the 3-Share instance of Zorro running in MM.

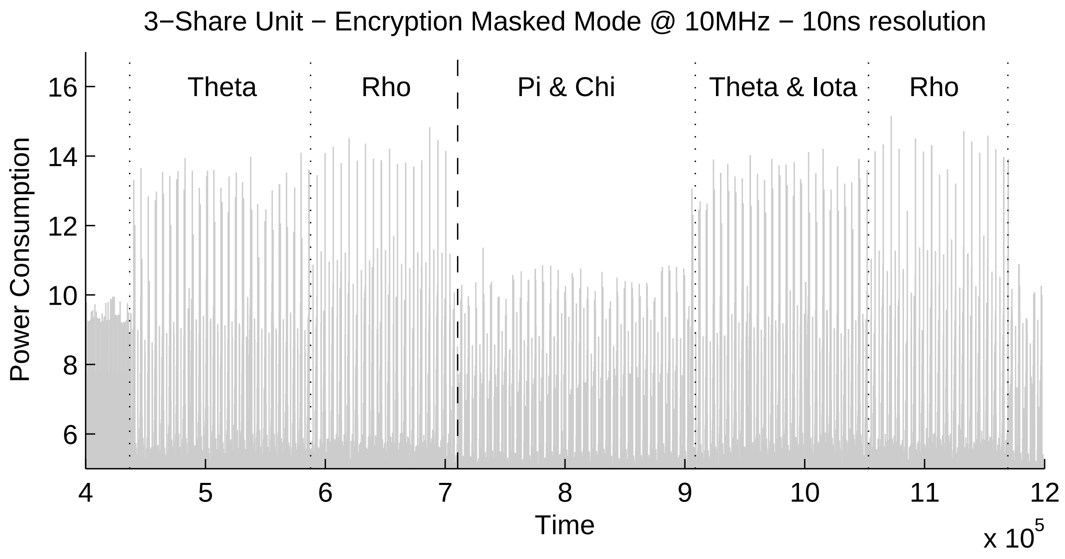


Figure 6.6.: Simulated power consumption during the first two rounds of the 3-Share instance of Zorro running in NM.

Figure 6.6 shows the first two rounds in detail. They are separated with a dashed line. A big difference exists between the iterative execution of $\theta\,\iota$ and ρ and the iterative execution of $\pi\,\chi$. This again has Secret Sharing (SS) issues. θ, ι, and ρ are only dependent of one share. They can be processed for each share sequentially. The loading is done like in NM. The whole register is shifted. If a function works with $n-1$ shares, it must be guaranteed that the implementation works only with $n-1$ shares at the same time in all concerns. In our case, we only work with one share during the loading of the information for χ and ι. We only shift the bits of the register which belong to one share during the loading process. These are only 48 bits while the whole register contains 256 bits. Therefore, the power consumption of the loading steps of χ and ι are reduced in comparison to the other operations. The different power consumption shapes because of the loading, like in the NM, are not present. It seems that the generally higher power consumption absorbs this effect.

6.3. Power Analysis against Zorro

The first analysis revealed that the register operations influence the power consumption in a high dimension. To exploit this fact, our first attack targets a storage operation of the *SubState* register. To keep the attack as simple as possible the numbers of targeted key bits are kept low. As a starting point an encryption of the 3-Share instance is taken. The key size is defined by 256 bits. The Authenticated Encryption (AE) system provided by Zorro gives the possibility to send header data concatenated with the key to the Application-Specific Integrated Circuit (ASIC). For this attack, the key is kept constant and the header data is changed per encryption. The header is used as non-constant value for the power analysis.

6.3.1. Detailed Structure of the Attack

To keep the key guess number as small as possible, the first round is attacked. In this round, Zorro computes only θ and ρ. The result of θ is a perfect target because each execution only works with the information of two slices. A slice consists of 25 bits. If the algorithm works with a key of 256 bits and they are stored in a linear lane based way, each slice contains 4 key bits. Attacking one θ execution, which works with two slices, results in an attack against 8 key bits. Thus, $2^8 = 256$ key guesses must be considered for the attack. This is a small amount and should not lead to a long computation time. For each attacked slice, 21 bits are left. Before the first round starts, they consist of 13 header bits and 8 state bits. The header bits can be chosen and hence they are known. The state bits are initialized with zeros for the NM. In MM, they are randomly initialized by the user. For the computation of the hypothetical values they are assumed as zero in all cases. As a distinguisher, we have chosen the Person correlation coefficient.

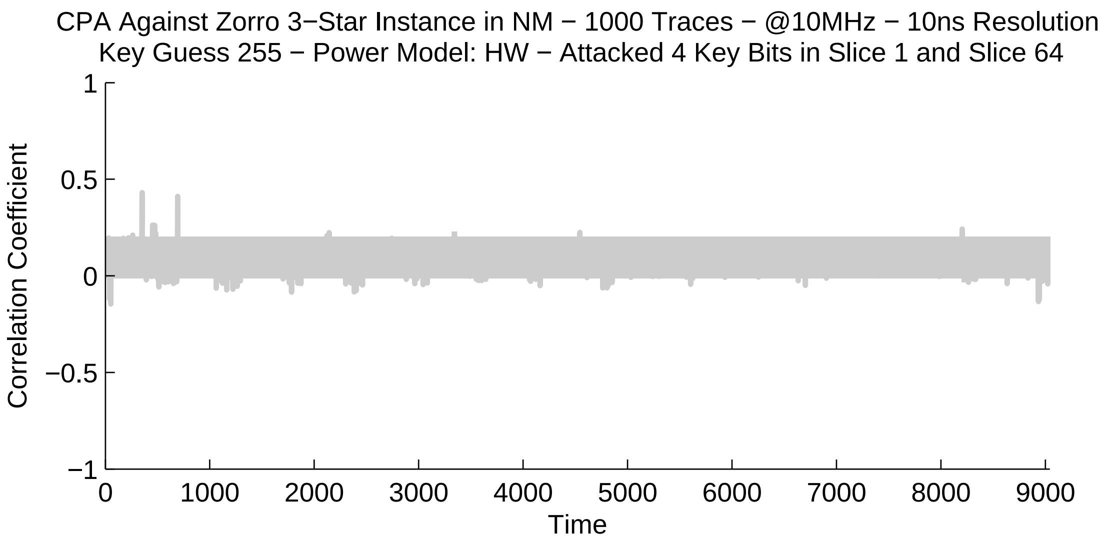


Figure 6.7.: CPA result for the correct key guess of ZORRO 3-Share running in NM.

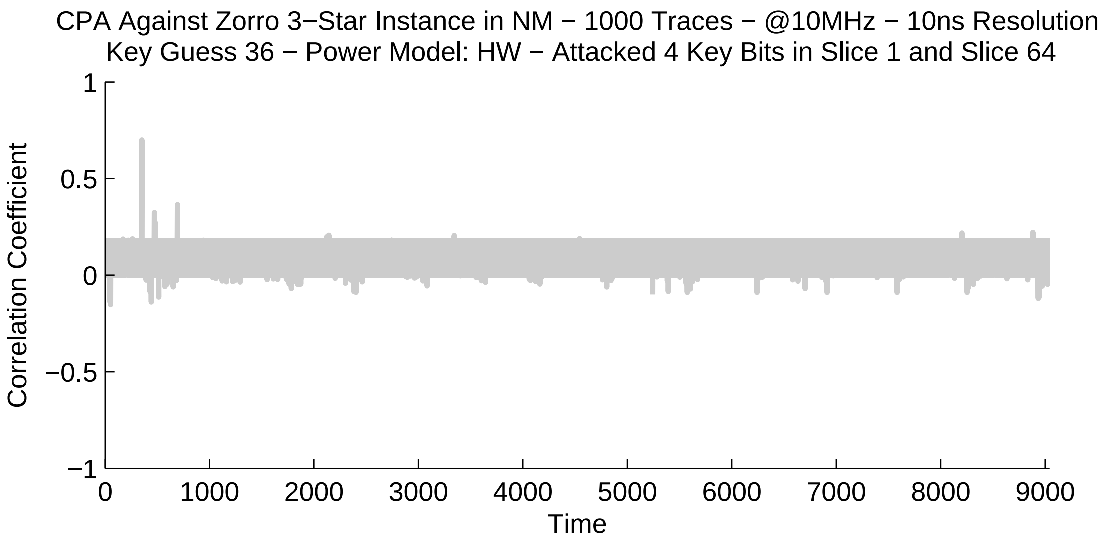


Figure 6.8.: CPA result for an incorrect key guess of ZORRO 3-Share running in NM.

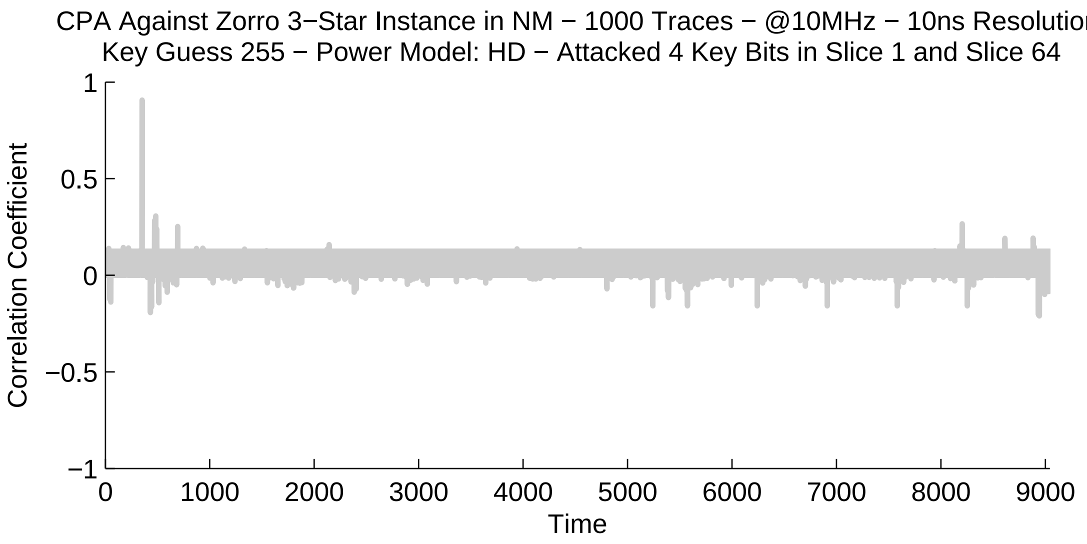


Figure 6.9.: CPA result for the correct key guess of Zorro 3-Share running in NM

6.3.2. CPA Results of Zorro with Normal Mode Enabled

At first, 1000 traces were simulated. To keep the simulation time short, only the attacked round was simulated. The first comparison was made with the Hamming weight as a power model. Figure 6.7 shows the result for the correct key guess of the attack against the first θ execution. The highest peak is at the beginning of the trace. Because we attacked the first execution of θ, this is the expected location in time. Unfortunately, incorrect key guesses lead to higher peaks (see Figure 6.8).

As a second attempt, we changed the power model to the Hamming distance. The updated values of θ get stored at the same place in the register where the old ones were stored. Consequently, the distance should lead to better results. The results can be seen in Figure 6.9 and 6.10, again for the correct and an incorrect key guess. It shows that the correlation coefficient for the correct key guess was significantly higher, providing a value of more than 0.9. Note that the confidence interval, where 99.99 % of all single points are located in the normal distribution model, is about 0.12 for 1000 traces. The attack can be therefore considered as successful since only the correct key guess leads to a significant peak lying outside this interval.

However, it also shows that other key guesses provide also significant peaks. In particular, 8 key guesses lead to the same output. The reason for this is the fact that θ is a linear function such that several key guesses lead to the same linear output. Therefore, this attack can reduce the key guesses from 256 down to only 8. A figure for the highest correlation coefficient for each key can be seen in Figure 6.11. This attack can be mounted against all θ executions. The result is the same for each slice. Consequently, the security of the system can be reduced from 256 bits down to 96 bits which is a huge

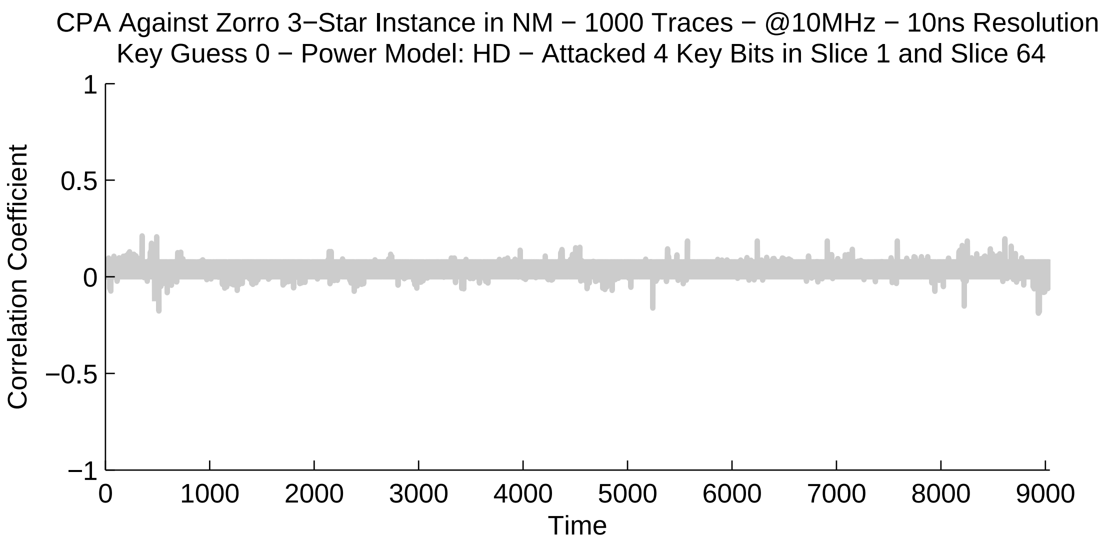


Figure 6.10.: CPA result for an incorrect key guess of Zorro 3-Share running in NM.

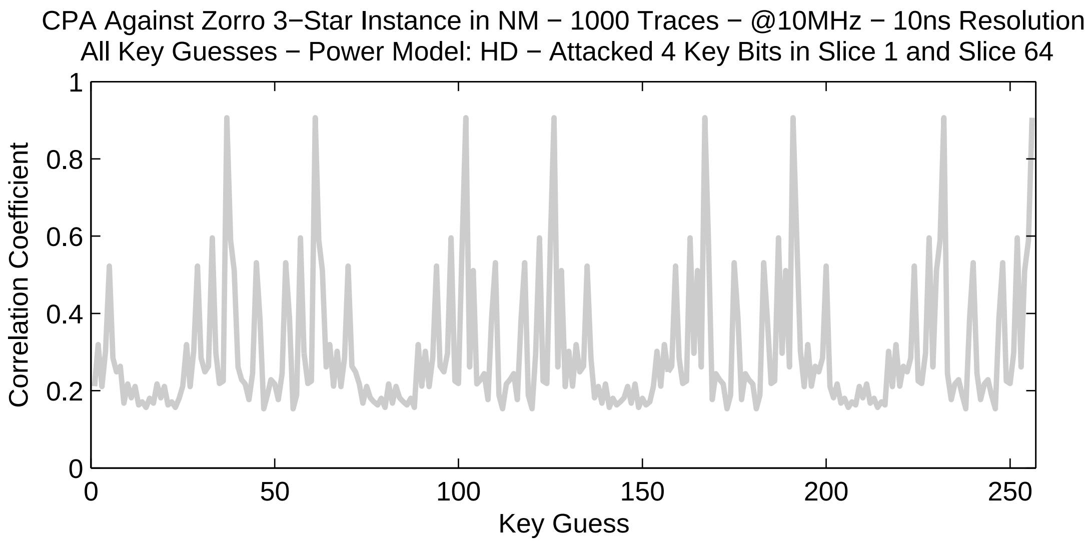


Figure 6.11.: Maximum correlation coefficients of the CPA result for all possible key guesses of Zorro 3-Share. It runs in NM.

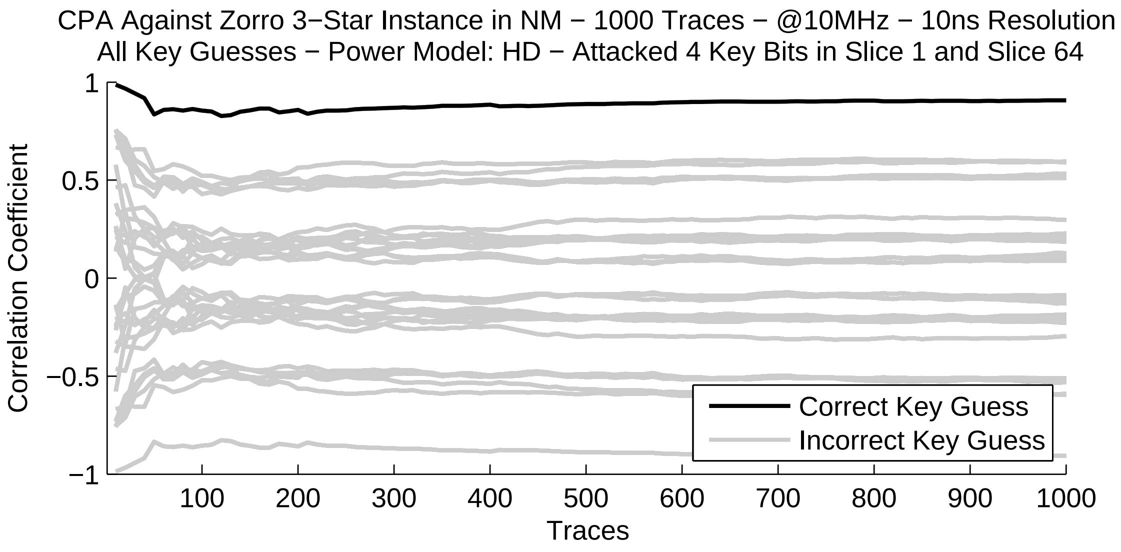


Figure 6.12.: CPA results for all possible key guesses of Zorro 3-Share running in NM. As point of time the point with the highest peak from Figure 6.10 was taken.

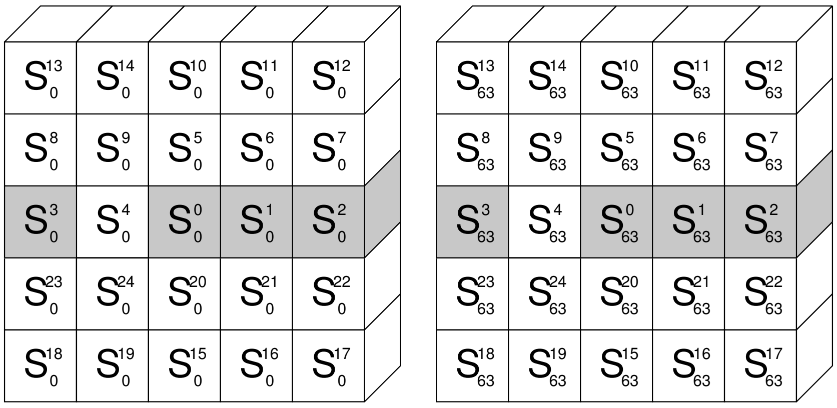
reduction. With this attack it was possible to attack the non-protected encryption of Zorro successfully.

We further analyzed how many traces are necessary to achieve a successful attack. For that reason, the point of time was kept constant at the point which leads to the highest correlation coefficient. Additionally, the number of traces was increased step by step while an attack was performed. The result is shown in Figure 6.12. It shows that a very small number of traces (under 100) leads already to a successful attack. This nicely corresponds to the theoretical estimation given in [21].

We now analyze θ in more detail. The function works with 2 slices. For the first execution of θ, they are s_0 and s_{63}. The result is stored at the same location where the old values were stored. The Hamming distance for the first bit is given in Equation (6.1).

$$HD_0 = \mathbf{s_0^0} \oplus \mathbf{s_0^0} \oplus s_0^4 \oplus s_0^9 \oplus s_0^{14} \oplus s_0^{19} \oplus s_1^{24} \oplus s_{64}^6 \oplus s_{64}^{11} \oplus s_{64}^{16} \oplus s_{64}^{21} \tag{6.1}$$

θ computes the sum of two rows of the slices. Because the Hamming distance adds the content of the old register value to the result, one value gets canceled (see bold values in Equation (6.1)). This is exactly the value that differs from row to row. Thus, each result computed for the same row is equal and only distances of the value 0, 5, 10, 15, 20, and 25 are possible. Figure 6.13 shows a graphic for the two slices used by θ. Only 8 bits are influenced by the key (highlighted in gray) and each bit that θ computes is only influenced by two key bits. Consequently, an equal sum of these bits results in an equal



S_0^{13} S_0^{14} S_0^{10} S_0^{11} S_0^{12}
S_0^{8} S_0^{9} S_0^{5} S_0^{6} S_0^{7}
S_0^{3} S_0^{4} S_0^{0} S_0^{1} S_0^{2}
S_0^{23} S_0^{24} S_0^{20} S_0^{21} S_0^{22}
S_0^{18} S_0^{19} S_0^{15} S_0^{16} S_0^{17}
S_{63}^{13} S_{63}^{14} S_{63}^{10} S_{63}^{11} S_{63}^{12}
S_{63}^{8} S_{63}^{9} S_{63}^{5} S_{63}^{6} S_{63}^{7}
S_{63}^{3} S_{63}^{4} S_{63}^{0} S_{63}^{1} S_{63}^{2}
S_{63}^{23} S_{63}^{24} S_{63}^{20} S_{63}^{21} S_{63}^{22}
S_{63}^{18} S_{63}^{19} S_{63}^{15} S_{63}^{16} S_{63}^{17}

Figure 6.13.: Naming convention for the two slices used by θ. The gray bits are key bits.

Hamming distance. This is the reason why different keys lead to the same coefficient in our attack. If only one key bit is different the distance only differs by the value of 5 which is a difference of 20 %. This is the reason why the coefficient between correct and incorrect key guesses sometimes only differs by 0.2. In Figure 6.12, the coefficients can be arranged in 6 different groups. This can be explained by to the 6 different Hamming distances θ provides.

6.3.3. CPA Results of Zorro with Hiding Mode Enabled

The next attack was mounted against Zorro 3-Share instance which runs in HM. The attack point is still the same, but we directly applied the Hamming distance power model. We reduced the dummy operations to a minimum to need a smaller quantity of traces for the attack. Only one dummy operation per permutation is performed. We simulated two rounds of the permutation. This guarantees that the round which works with the right values is simulated. In HM, 8 different starting positions are possible because of the shuffling. This leads to a randomization degree of $(\#DummyOps + 1) * (\#ShufflePos) = (1 + 1) * 8 = 16$. Thus, altogether the correlation coefficient should by reduced to approximately $\frac{1}{16}$. This leads to a theoretical correlation coefficient under 0.1.

The first experiment was made with 1000 traces. The result for the correct key guess can be seen in Figure 6.14. It shows that 1000 traces are too few for a successful attack. Remember, with hiding the amount of traces gets squared. The first attack needed around 50 traces to be successful. Therefore, this attack would need around $16^2 \times 50 = 12800$ traces to be successful. Due to a lack in time, we only simulated 5000 traces. The result for the correct key guess can be seen in Figure 6.15. There is no significant peak at the beginning of the rounds and we cannot distinguish between correct and incorrect key guesses. Altogether, the attack was not successful and a reduction of the key space was not possible.

Note that in the second half of each permutation, negative or positive peaks are observ-

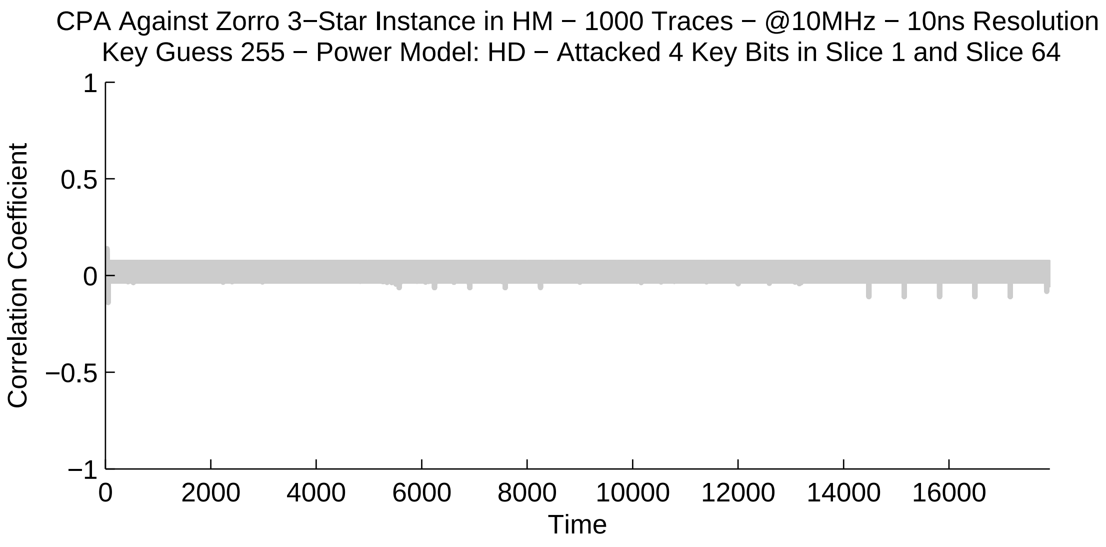


Figure 6.14.: CPA result using 1000 trace for the correct key guess of Zorro 3-Share running in HM.

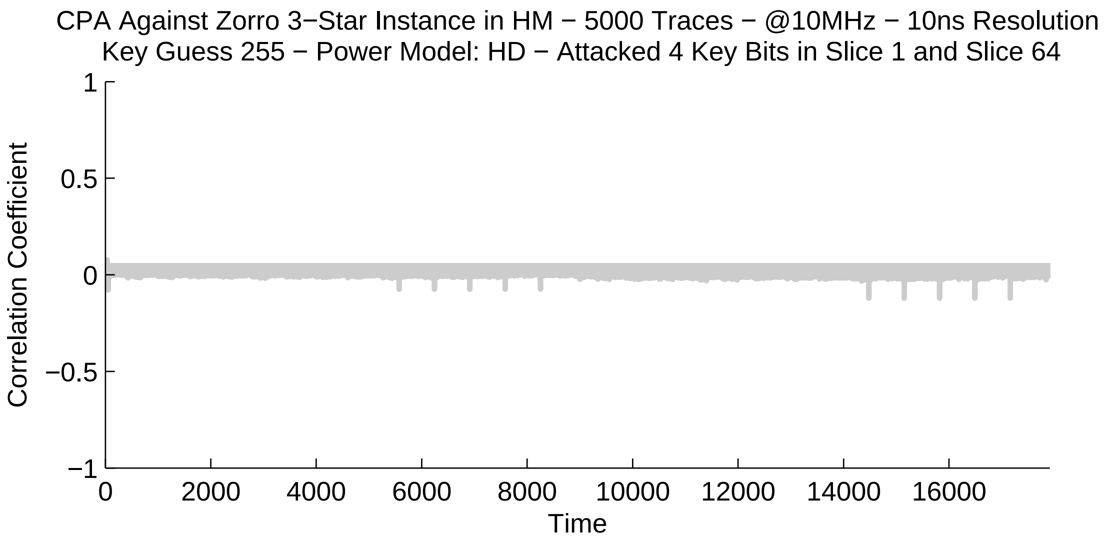


Figure 6.15.: CPA result using 5000 traces for the correct key guess of Zorro 3-Share running in HM.

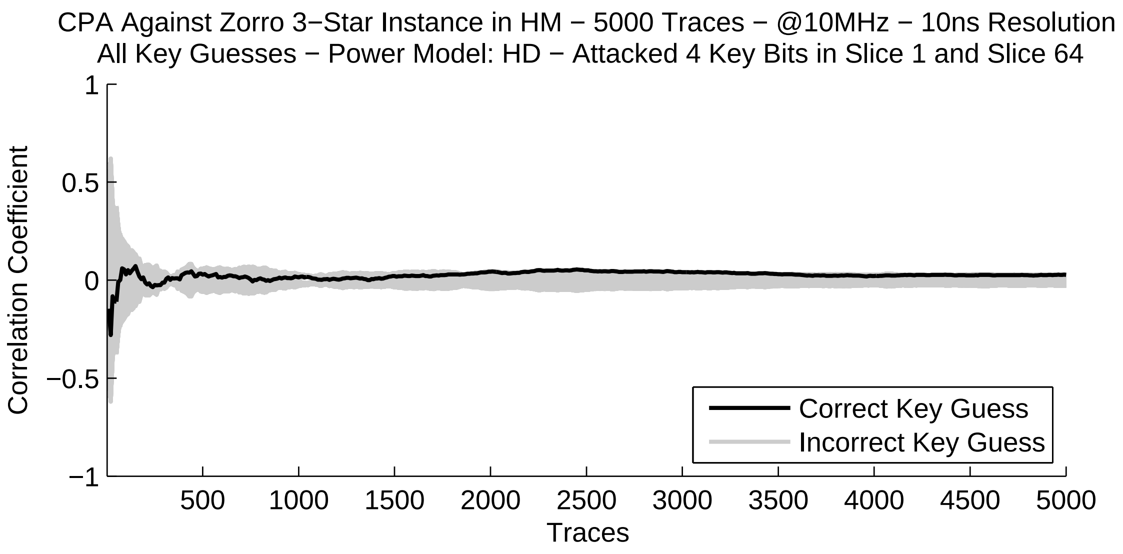


Figure 6.16.: CPA results for all possible key guesses of Zorro 3-Share running in HM.

able. These peaks can be found in the results for the NM as well, but they are not that significant because they are hidden in noise. They are due to the reset of the *SubState* register after ρ. After ρ, the register contains parts of the result of θ. This is because the result of ρ is not stored back into the register. Thus, a small correlation can happen because of the reset. In HM, we shuffle the loading, the saving, and the execution steps. The reset of the intermediate register stays the same. The noise around these peaks is reduced because of the shuffling, the peak itself is not. That is the reason why they remained nearly the same as in NM.

In addition, a plot for the increasing amount of traces was made. The result can be found in Figure 6.16. As point of time the precalculated point with the highest peak was taken. It can be seen that the correct guess has nearly the highest peak after about 1500 traces. But the difference between a correct and an incorrect key guess is too small for a distinction. From the last attack, we learned that the smallest difference is only 0.2. Therefore, it should be decreased in this case to $0.2 * \frac{1}{16} = 0.0125$.

6.3.4. CPA Results of Zorro with Masked Mode Enabled

We performed a CPA attack Zorro 3-Share instance in MM. For this example, we used 5000 traces. As power model, the Hamming distance was taken. For this attack, we simulated only one round. The CPA result for the correct key guess can be found in Figure 6.17. As expected, there is no significant peak observable. In comparison to the attack results for the HM, the coefficient looks more random and the peaks in the last half of the round disappeared. In MM, Zorro computes the results based on a

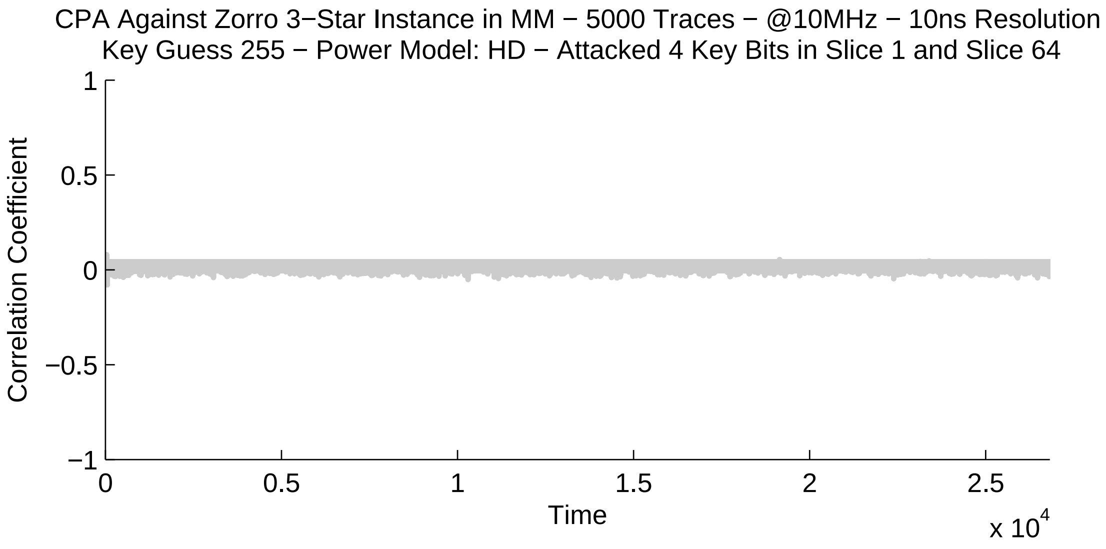


Figure 6.17.: CPA result for the correct key guess of Zorro 3-Share running in MM.

random initialization. At no time correct intermediate values are present. Therefore, even the reset of the *SubState* is not dependent on parts of the real results. This leads to a decreasing correlation coefficient with an increasing amount of traces.

To complete the analysis, Figure 6.18 shows CPA results for on increasing number of traces for all key guesses. It shows that the attack was not successful with the given amount of power traces.

6.3.5. Final Notes on the Attacks

We tried to attack the implementations with a very generic approach. The only knowledge needed for the attack concerns the θ execution. We expected that the old register values get overwritten with the new results. This is a typical approach to update registers and thus can by guessed by the attacker easily. In addition, an iterative execution was assumed. In low-area implementations, this is a basic way to calculate functions.

In order to evaluate the attacks against HM and MM in a more meaningful way, more traces must be considered for the attacks. A simulation of a higher number of traces takes too much time. An advantage of power analysis mounted against a real device is that the measurement of up to a million traces is possible in a feasible time. Therefore, to evaluate the countermeasures more meaningfully, attacks against the fabricated ASIC must be mounted.

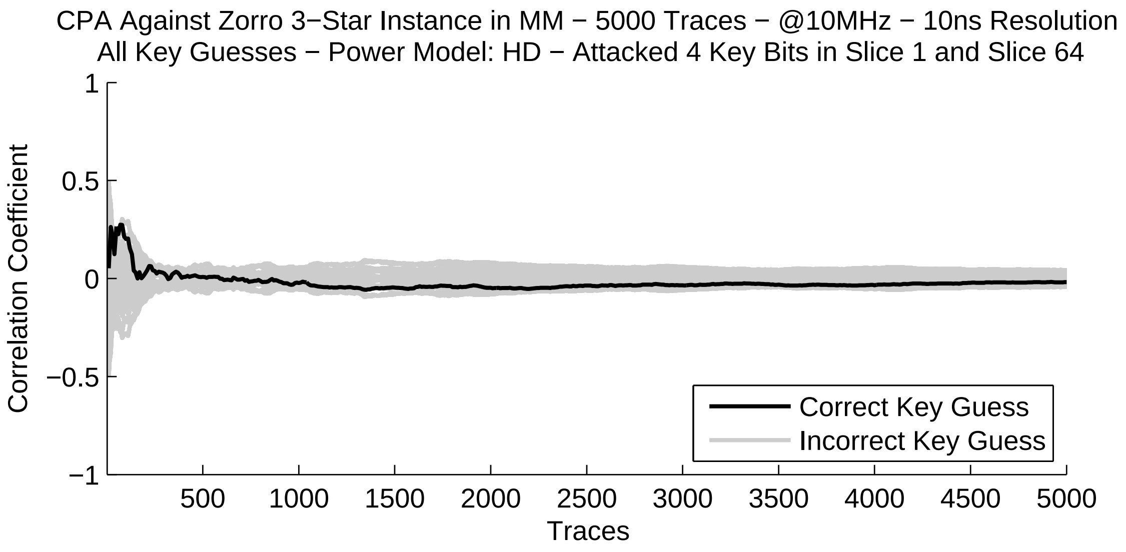


Figure 6.18.: CPA results for all possible key guesses of ZORRO 3-Share running in MM.

Chapter 7

Hardware Architecture of FastKeccak

Recently, the amount of sensitive digital data increased steadily. In order to process this amount of confidential data securely and in a reasonable time, throughput-trimmed implementations of cryptographic algorithms are required.

In this chapter, we want to introduce our high-throughput implementations, called FastKeccak, of an Authenticated Encryption (AE) system based on Keccak. All implementations were synthesized for the *Xilinx Kintex-7* and the *Xilinx Virtex-7* Field Programmable Gate Array (FPGA).

The chapter is structured as follows. In Section 7.1, we introduce our hardware architectures. Section 7.2 outlines the results for different FPGA boards. In Section 7.3, the results are compared with related work concerning Keccak. Finally, in Section 7.4, we compare the sponge-based results with block-cipher implementations.

7.1. The Hardware Architecture

A common approach to speed up block-cipher based algorithms is to pipeline them or to use multiple instances of the whole implementation. Figure 7.1 shows a structure using both the pipeline- and the multiple-instance approach. Most of the time the first chosen approach is to pipeline the implementation. Additional registers are used to speed up the algorithm. Very often a positive area-time relation can be gained from such approaches. If a higher throughput is needed, instances are often parallelized. Now the area-time relation stays the same, but the throughput increases.

Nevertheless, when it comes to sponge-based primitives, they have one in common: the result of input block $n-1$ influences block n. Consequently, a block must be fully processed before a new one can be absorbed. This leads to a drawback in speeding up an implementation. Pipeline approaches are not realizable. In addition, multiple instances cannot encrypt parts of the same massage. However, a speedup can be gained from a faster permutation. Additionally, a higher block size and thus a lower security level can increase the throughput.

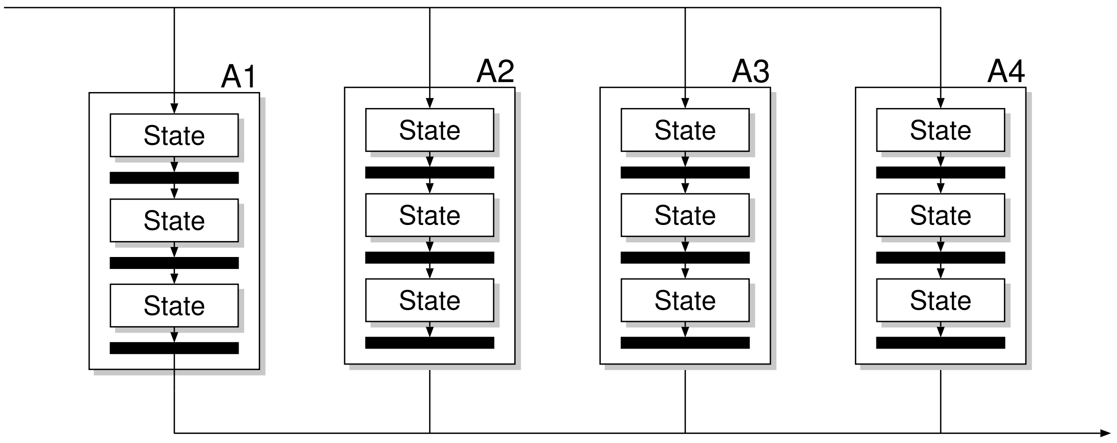


Figure 7.1.: Standard approach to maximize the throughput of block-cipher based algorithms.

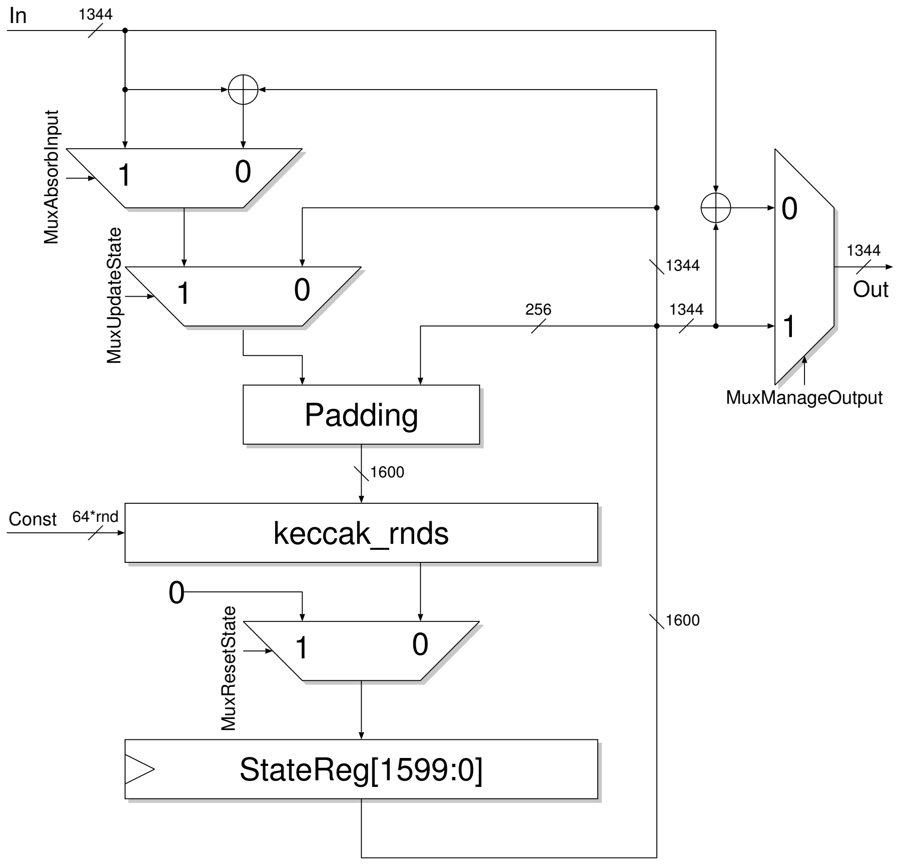


Figure 7.2.: Datapath of the FastKECCAK architecture.

We work with KECCAK-f[1600]. Our approach is to fully parallelize the permutation. This leads to 1600 instances of θ and 320 instances of χ. π and ρ are only rewiring processes when it comes to a fully parallelized architecture. ι still keeps the same and can be applied with a 64-bit XOR operation. The constants for it are stored in a Look-up Table (LUT). The controlling is made by a simple Finite State Machine (FSM). The block size is defined by 1344 bits and thus the security level is given by 128 bits. These alterations lead to a KECCAK round which can be computed in a single cycle.

In order to ensure a correct computation, additional multiplexers around the permutation are necessary. If several KECCAK round instances are concatenated, the critical path increases. But the path is only increased because of these additional KECCAK round instances. The structure around the permutation and the FSM stays the same. Consider that only one KECCAK round is used. In our case, around 50 % of the critical path is needed because of the KECCAK round, the rest belongs to the FSM and the surrounding multiplexers. KECCAK-f[1600] works with 24 rounds. Datapaths with 1, 2, 3, 4, 6, 8, 12, and 24 KECCAK round instances can exploit this fact and increase the throughput.

Therefore, 8 different architectures are possible by unrolling the permutation. The used KECCAK round instance stays the same and is only used multiple times. Now the LUT which is responsible for the ι constants must provide 64 bits per used KECCAK round and cycle.

Figure 7.2 shows an overview of the datapath of our system. As storage element we used a register of 1600 bits. The multiplexer in front of it is responsible to initialize the state. The *keccak_rnds* entity can be adjusted to provide different numbers of KECCAK round instances. A detailed view of the entity can be seen in Figure 7.3. The different function of KECCAK are concatenated. Each function can compute the result of the whole state in a single cycle.

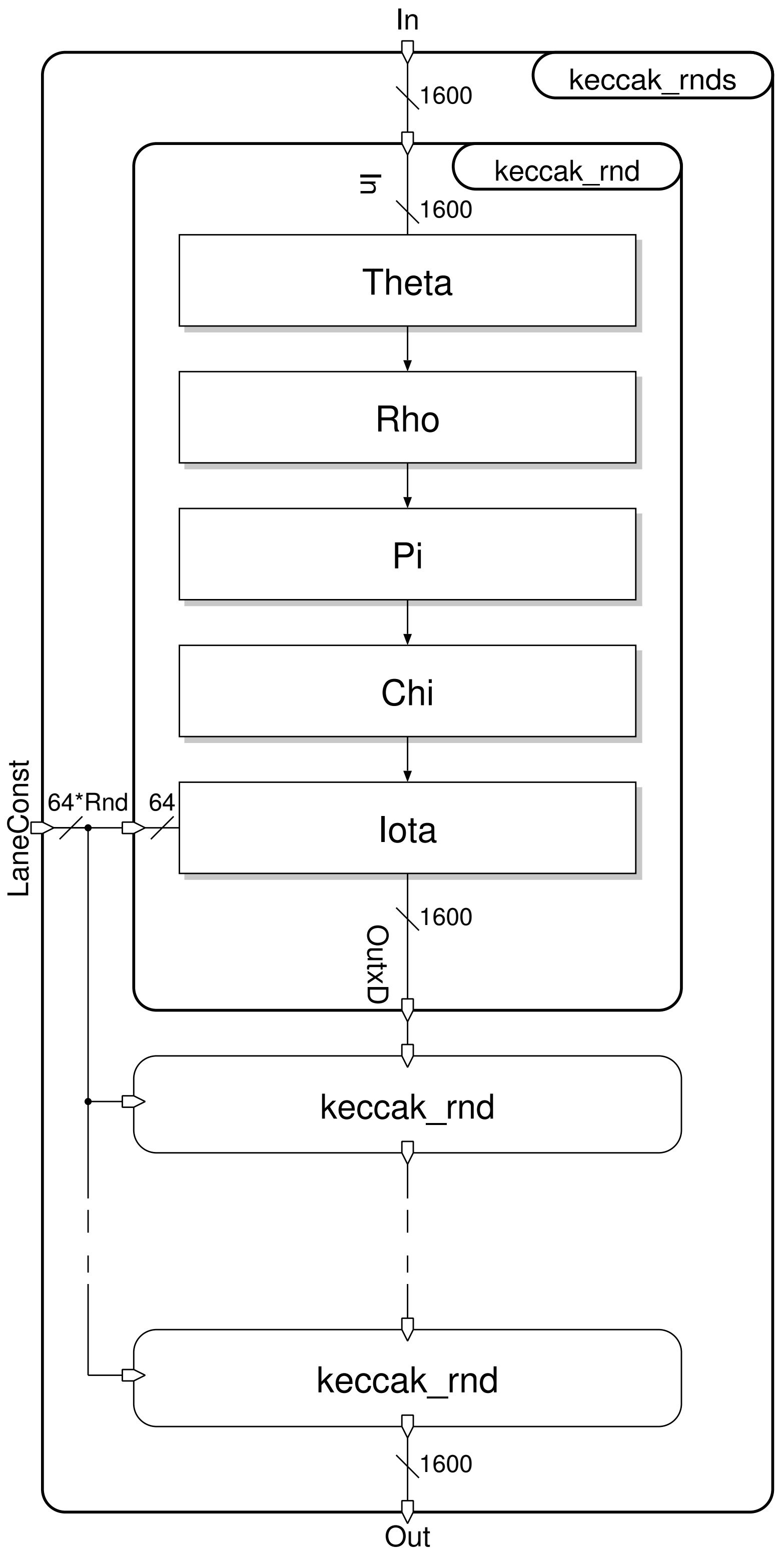


Figure 7.3.: Round instances of the FastKECCAK architecture.

7.2. FPGA Results

The implementations were made with Very High Speed Integrated Circuit Hardware Description Language (VHDL). For synthesis *Xilinx Vivado Design Suite 2013.1* was used. The block size is the same for all architectures and is given by 1344 bits. Our first targeted platform was the *Xilinx Kintex-7 FPGA KC705* evaluation board. Table 7.1 shows post synthesis results for the different architectures. The first column contains the number of used KECCAK round instances. In order to distinguish between the different implementations, the notation #x, where x denotes the number of concatenated KECCAK round instances, is used. The second column represents the needed Flip-Flops (FFs). 1600 are used for the state of KECCAK. The rest belongs to the FSM. They are used as counter for the round iterations and as an FSM counter. The more KECCAK round instances are used the fewer iterations are performed. Therefore, the number of counter FFs for the round iterations decrease. The third column shows the needed LUT count. It increases nearly linear with the used KECCAK round instances. The throughput is given in column five. Between #6 and #8 and #12 and #24 a decrease of the throughput can be seen. The reason for that is additional routing. Because of the higher amount of combinatorial units, the routing complexity increases. In the case of #6 and #8 and #12 and #24, the increase of the critical path because of the additional routing is higher than the decrease due to the additional KECCAK round instances and therefore the throughput decreases. Figure 7.4 shows the throughput results for all architectures graphically. An area-throughput plot for all 8 architectures can be found in Figure 7.5. A continuous decrease can be seen. This has the following reason. The relative influence on the critical path because of the round instances increases from additional instance to instance. Therefore, the relative influence on the critical path because of the FSM and the multiplexers decreases. But the size of the implementation increases in a linear way with the round instance count. Altogether this leads to a higher, but not linearly increasing throughput and the area-throughput relation decreases.

The same architectures were synthesized for the *Xilinx Virtex-7* FPGA. The post

Table 7.1.: Kintex-7 XC7K325T-2FFG900C FPGA results of FastKECCAK.

Inst. [#]	**FFs** [#]	**LUTs** [#]	**Frequency** [MHz]	**Throughput** [Gbit/s]
1	1607	5014	301.5	16.88
2	1607	7321	197.0	22.06
3	1606	9440	145.9	24.51
4	1605	11943	113.9	25.52
6	1605	16154	82.2	27.62
8	1604	21119	61.8	27.69
12	1604	29582	44.6	29.94
24	1602	57826	21.9	29.36

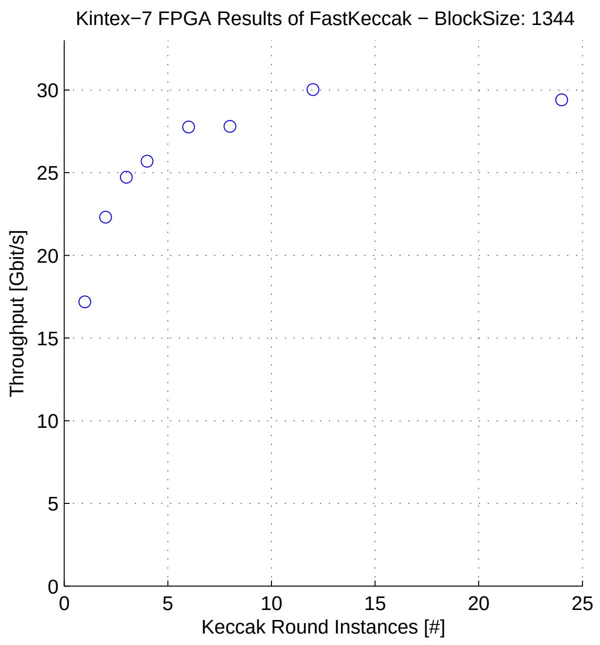


Figure 7.4.: Throughput results of FastKeccak for the Kintex-7 XC7K325T-2FFG900C FPGA.

synthesis for all 8 architectures can be seen in table Table 7.2. The same results are displayed in Figure 7.6 graphically. A general speedup in comparison to the previous results can be seen. This can be explained by a higher *speed grade* of this FPGA. In addition, the throughput increase between #6 and #8 and #12 and #24 is better. This apparently can be refereed to the size of the FPGA. Because of the higher amount of combinatorial units the routing is performed in a better way. This assumption is confirmed by looking at the critical path caused by the routing. In these cases, the critical path caused by the routing is about 0.5 % shorter than for the *Xilinx Kintex-7*. An area-throughput plot can be found in Figure 7.7. The decrease of the area-throughput relation can be traced back to the same reason as for the *Xilinx Kintex-7* FPGA.

7.3. Comparison with Related Work

Because of the bad area-throughput results Keccak-*f*[1600]-based primitives are often not unrolled to a factor of 24. In addition, most of the time a block size of 1088 bits or 1024 bits is used and comparisons between throughputs of different FPGAs are not that easy to interpret. But to the best of our knowledge, the fastest Keccak implementation was reported by Homsirikamol et al. [15] and has a throughput of 13.0 Gbit/s. They used an *Altera StarixIV FPGA*. As block size 1088 bits were taken.

Furthermore, Bertoni et al. [6] reported an implementation for a FPGA with a through-

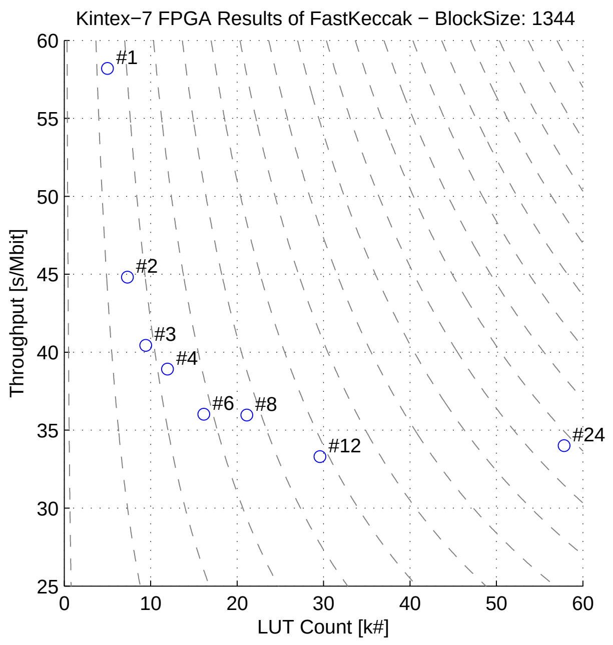


Figure 7.5.: LUT count in relation to the throughput of FastKeccak for the Kintex-7 XC7K325T-2FFG900C FPGA.

put of 8.7 Gbit/s. An *Altera Stratix III* was used. The block size was given by 1024 bits.

Another Keccak-*f*[1600] FPGA implementation was reported by AIST RCIS [1]. A *Xilinx Virtex-5* FPGA was used. The throughput is given by 8.40 Gbit/s. One input block has the size of 1024 bits.

Strömbergson [36] reported an implementation for the *Altera Stratix III* FPGA. A throughput of 10 Gbit/s was presented. The block size is given by 1024 bits.

Baldwin et al. [3] implemented an architecture of Keccak-*f*[1600] with a block size of 1088. They used a *Xilinx Virtex-5* FPGA and reported a throughput of 6.27 Gbit/s.

Our fastest implementation provides a throughput of over 34 Gbit/s. This is a throughput increase of more than 200 % compared with the implementation presented in [15]. Therefore, it is even mentionable although different block sizes and FPGAs were used.

A total other approach was reported by Homsirikamol et al. [15]. Their implementation was pipelined to hash different messages at the same time. As throughput they reported 19.2 Gbit/s. Such an approach is not that important for encryption systems because in many applications the speed for one message in significant. But our implementation is even faster than this multi-hash implementation. Therefore, this is also mentionable.

Summarized, we are able to report the fastest FPGA implementation of an AE system based on Keccak-*f*[1600]. Detailed results of related work compared with ours can be found in Table 7.4.

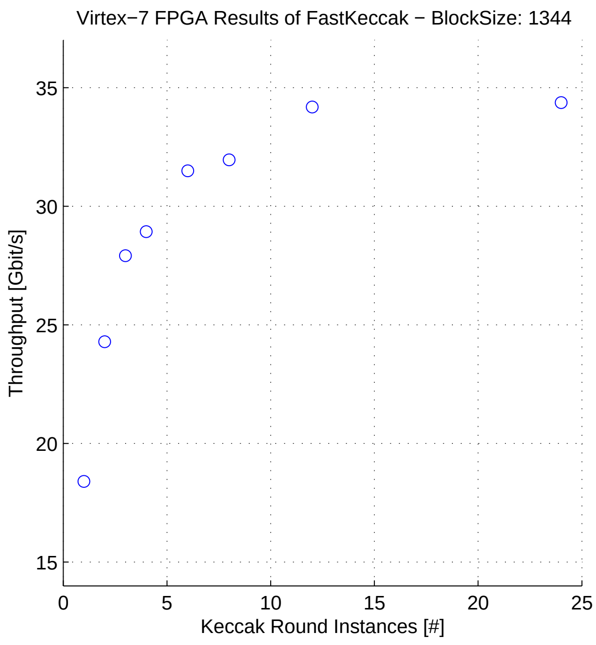


Figure 7.6.: Throughput results of FastKECCAK for the Virtex-7 XC7VX485T-2FFG1761C FPGA.

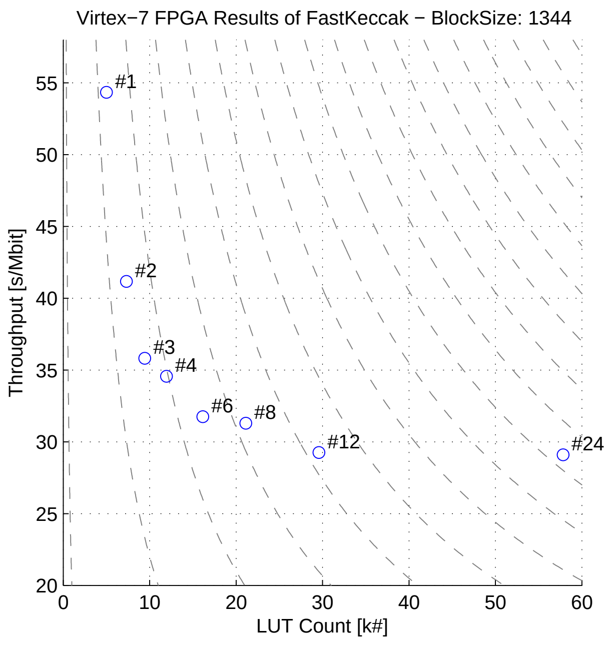


Figure 7.7.: LUT count in relation to the throughput of FastKECCAK for the Virtex-7 XC7VX485T-2FFG1761C FPGA.

7.4. A Sponge against a Block

Up to now, most of the time block-cipher based algorithms are used for AE systems. Very famous modes of operations are called Offset Codebook Mode (OCB) and Galois/Counter Mode (GCM). Muehlberghuber et al. [25] presented Advanced Encryption Standard (AES) and Serpent implementations for these modes. They used an *Altera Stratix IV GT* FPGA and presented for all architectures a throughput higher than 100 Gbit/s. Our sponge-based system is around 3 times slower. This is due to the fact that pipeline and multi-instance approaches are not suitable for our primitive. Detailed results of the different implementations of block-cipher and sponge-based AE systems can be seen in Table 7.3.

Table 7.2.: Virtex-7 XC7VX485T-2FFG1761C FPGA results of FastKeccak.

Inst. [#]	**FFs** [#]	**LUTs** [#]	**Frequency** [MHz]	**Throughput** [Gbit/s]
1	1607	5013	328.6	18.40
2	1607	7322	216.9	24.28
3	1606	9440	166.2	27.91
4	1605	11943	129.1	28.92
6	1605	16154	93.7	31.49
8	1604	21119	71.3	31.95
12	1604	29582	50.9	34.18
24	1602	57826	25.6	34.36

Table 7.3.: AE system results for block-cipher and sponge-based primitives.

Source	**Device**	**Algorithm**	**Mode**	**TP** [Gbit/s]
Muehlberghuber et al. [25]	*Altera Stratix IV GT*	AES	OCB	128.00
Muehlberghuber et al. [25]	*Altera Stratix IV GT*	AES	GCM	105.00
Muehlberghuber et al. [25]	*Altera Stratix IV GT*	Serpent	OCB	141.00
Muehlberghuber et al. [25]	*Altera Stratix IV GT*	Serpent	GCM	104.00
	This work			
FastKeccak-12	*Xilinx Kintex-7 XC7K325T2FFG900C*	Keccak-f[1600]	-	29.94
FastKeccak-24	*Xilinx Virtex-7 XC7VX485T2FFG1761C*	Keccak-f[1600]	-	34.36

Table 7.4.: Related work and FastKECCAK results.

Source	**Device**	**Logic** [#]	**FFs** [#]	**Block Size** [Bits]	**Max. Freq.** [MHz]	**Throughput** [Gbit/s]
Bertoni et al. [6]	*Altera Stratix III EP3SE50F484C2*	4684 ALUTs	2641	1024	206	8.70
Bertoni et al. [6]	*Xilinx Virtex-5 XC5VLX50FF324-3*	1330 slices	2641	1024	122	5.20
Homsirikamol et al. [15]	*Xilinx Virtex-5*	1272 CLB slices	-	1088	-	12.82
Homsirikamol et al. [15]	*Xilinx Virtex-6*	1207 CLB slices	-	1088	-	12.98
Homsirikamol et al. [15]	*Altera Stratix III*	4213 ALUTs	-	1088	-	12.40
Homsirikamol et al. [15]	*Altera Stratix IV*	4129 ALUTs	-	1088	-	12.97
Strömbergson [36]	*Altera Stratix III EP3SE50F484C2*	4550 ALUTs	2670	1024	176	10.00
Strömbergson [36]	*Xilinx Virtex-5 XC5VLX50FF324-3*	1483 slices	2669	1024	176	6.70
AIST RCIS [1]	*Xilinx Virtex-5 XC5VLX30*	1433 slices	2666	1024	205	8.40
Baldwin et al.[3]	*Xilinx Virtex-5*	1117 slices	-	1088	196	6.27
		This work				
FastKECCAK-12	*Xilinx Kintex-7 XC7K325T2FFG900C*	29582 LUTs	1604	1344	44	**29.94**
FastKECCAK-1	*Xilinx Kintex-7 XC7K325T2FFG900C*	5014 LUTs	1607	1344	301	**16.88**
FastKECCAK-24	*Xilinx Virtex-7 XC7VX485T2FFG1761C*	57826 LUTs	1602	1344	22	**34.36**
FastKECCAK-1	*Xilinx Virtex-7 XC7VX485T2FFG1761C*	5014 LUTs	1607	1344	25	**18.40**

Chapter 8

Conclusion and Future Work

This work dealt with implementations of an Authenticated Encryption (AE) system based on the permutation of KECCAK-f[1600]. First, a low-area design was the target. This design is secured with countermeasures against power-analysis attacks. The implemented countermeasures are called *hiding* and *masking*. Until now, 3 different masking schemes are available concerning KECCAK. We decided to combine all schemes on the resulting taped-out Application-Specific Integrated Circuit (ASIC), called ZORRO. Each scheme is implemented as an independent instance. The different instances are fully discrete AE systems and differ only with regard to the applied masking scheme. Due to the fact that all instances are combined on the same ASIC, their physical properties are rarely different. Therefore, ZORRO serves as an ideal evaluation platform for the different implemented countermeasures. In addition, the quantity of the performed dummy operations is selectable by the user. Up to 15 dummy operations are possible. This increases the evaluation opportunities of the ASIC furthermore. As a result, we can present masking scheme implementations more than 50 % smaller compared to related work. Hence, we can introduce the smallest AE systems based on KECCAK-f[1600] that is secured with masking and hiding countermeasures. Besides, we presented a detailed power analysis of an encryption followed by a Correlation Power Analysis (CPA) against one instance. The attack was mounted on different encryptions (non-protected encryption, encryption protected with hiding, and encryption protected with masking). With the attack against the non-protected encryption, we were able to reduce the security of the system from 256 bits down to 96 bits. All power analyses were performed based on the simulated power consumption of the ASIC. Since, our design was also taped out, the next task is to attack the fabricated ASIC. Afterwards, a comparison between the simulation-based results and the real power analysis can be made. This comparison can be used to interpret simulation-based results of further power analysis better.

The second target was a high-throughput implementation of the AE system. High-throughput trimmed implementations are mostly needed in environments which are not accessible by a public domain. In addition, countermeasures against power analyses reduce the computation speed. Therefore, this implementation does not provide additional security against power analysis. The design was synthesized for both, a *Xilinx Virtex-7* Field Programmable Gate Array (FPGA) and a *Xilinx Kintex-7* FPGA. The highest throughput is given by 34.36 Gbit/s and 29.94 Gbit/s, respectively. To the best of our knowledge this is the fastest FPGA AE system implementation based on KECCAK-f[1600].

Appendix A

ZORRO- ASIC Datasheet

A.1. Features

- Technology
  - United Microelectronics Corporation (UMC) 180 nm
- Cryptographic performance[1]

– 3-Share instance (norm/masked):	21888/113184 Cycles
– 3-Share-Star instance (norm/masked):	21888/113184 Cycles
– 4-Share instance (norm/masked):	21888/149640 Cycles

- Area

– 3-Share instance:	14000 Gate Equivalent (GE)
– 3-Share-Star instance:	14500 GE
– 4-Share instance:	17000 GE
– Altogether:	46000 GE

- Estimated power consumption

– 3-Share instance:	17.29 mW
– 3-Share-Star instance:	19.69 mW

[1]24 KECCAK-f[1600] rounds

  - 4-Share instance: 20.80 mW

- Package
  - QFN 56 (8x8)
- Maximum operating speed
  - 200 MHz
- Pads
  - 8-pad padframe power supply pins
  - 8-pad core power supply pins
  - 40 Input/Output (I/O) pins
- Supply Voltages
  - 3.3 V pad supply
  - 1.8 V core supply

A.2. Applications

Zorro is and AE system based on the Keccak permutation and can run in several modes:

- **Normal Mode (NM)**: The en-/decryption is performed without countermeasures.
- **Masked Mode (MM)**: The operation is secured with secret sharing.
- **Hiding Mode (HM)**: The calculation is performed in a normal way but there are additional hiding operations performed.
- **Secure Masked Mode (SMM)**: Secret sharing and hiding operations are combined.

A.3. Packaging

As package a QFN65 package is used. This package provides $8x8 = 56$ pins.

A.4. Bonding Diagram

The ponding of Zorro can be seen in Figure A.1.

A.5. Pinout

The pinout of Zorro can be seen in Figure A.2.

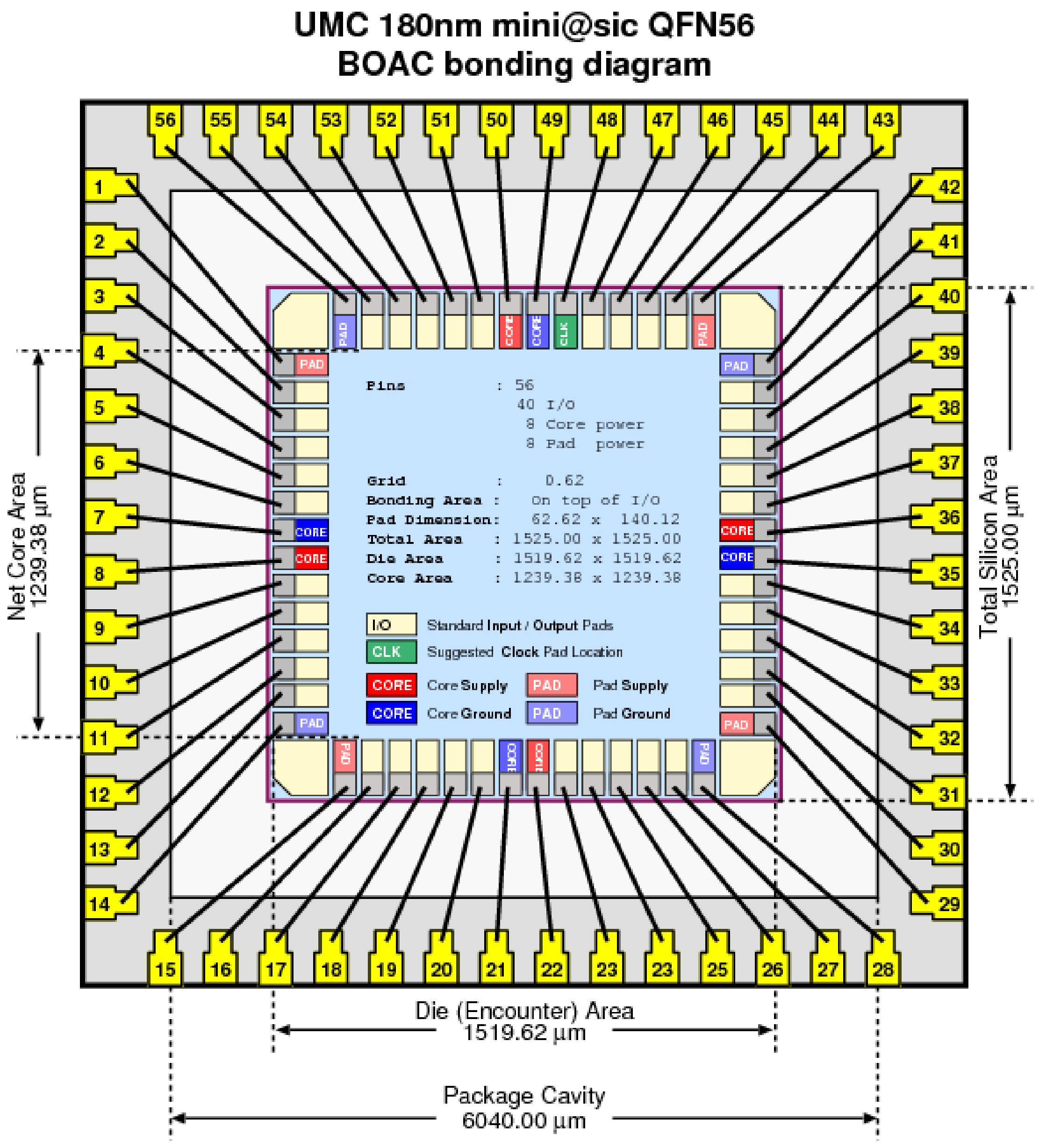

UMC 180nm mini@sic QFN56
BOAC bonding diagram
Pins : 56
40 I/O
8 Core power
8 Pad power
Grid : 0.62
Bonding Area : On top of I/O
Pad Dimension: 62.62 x 140.12
Total Area : 1525.00 x 1525.00
Die Area : 1519.62 x 1519.62
Core Area : 1239.38 x 1239.38
I/O Standard Input / Output Pads
CLK Suggested Clock Pad Location
CORE Core Supply
PAD Pad Supply
CORE Core Ground
PAD Pad Ground
Net Core Area 1239.38 µm
Total Silicon Area 1525.00 µm
Die (Encounter) Area 1519.62 µm
Package Cavity 6040.00 µm

Figure A.1.: The bonding used by Zorro.

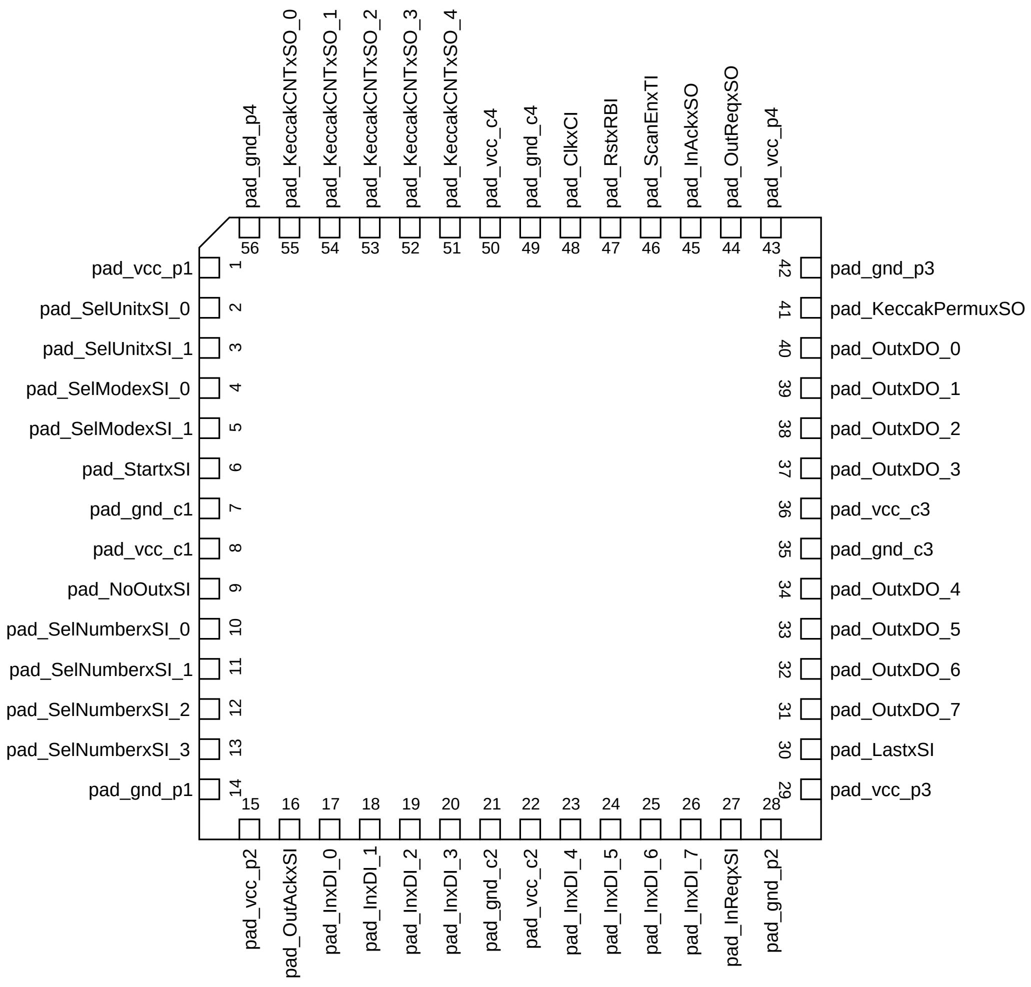


Figure A.2.: Pinout of ZORRO.

A.6. Pin Description

A detailed pin description can be found in Table A.1

A.7. Interface Description

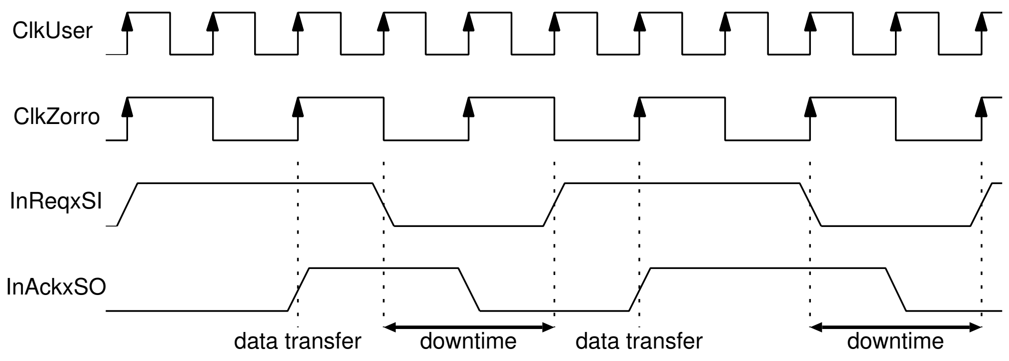


Figure A.3.: Sequence diagram for sending data from the user to ZORRO.

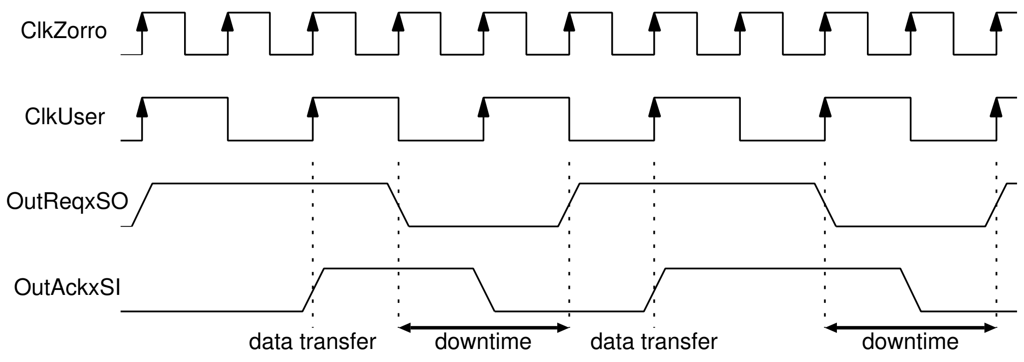


Figure A.4.: Sequence diagram for sending data from ZORRO to the user.

As communication interface ZORRO uses a four way handshake protocol. ZORRO expects the input in byte chunks. The sequence of the communication interface protocol can be seen in Figure A.3 and Figure A.4.

Table A.1.: Pin description of Zorro.

Pad name	**Description**
pad_vcc_p1, pad_vcc_p2 pad_vcc_p3, pad_vcc_p4 pad_gnd_p1, pad_gnd_p2 pad_gnd_p3, pad_gnd_p4	Power supply for the padframe (3.3 V)
pad_vcc_c1, pad_vcc_c2 pad_vcc_c3, pad_vcc_c4 pad_gnd_c1, pad_gnd_c2 pad_gnd_c3, pad_gnd_c4	Power supply for the core (1.8 V)
pad_OutAckxSI, pad_InAckxSO	Acknowledge signal of the user and the chip
pad_OutReqxSO, pad_InReqxSI	Request signal of the chip and the user
pad_KeccakPermuxSO	High if chip performs the Keccak permutation
pad_KeccakCNTxSO[4:0]	Outputs the actual round of the permutation
pad_SelModexSI[1:0]	x0 ⇒ Encryption x1 ⇒ Decryption 0x ⇒ Normal Mode 1x ⇒ Masked Mode
pad_SelUnitxSI[1:0]	00 ⇒ 3-Share instance 01 ⇒ 3-Share-Star instance 10 ⇒ 4-Share instance 11 ⇒ RAM Testmode
pad_SelNumberxSI[3:0]	Quantity of dummy operations 0 <⇒ Hiding mode
pad_ClkxCI	Clock Input
pad_RstxRBI	Asynchronous Reset Input (Active low)
pad_ScanEnxTI	Signal to run the chip in ScanChain-Testmode
pad_StartxSI	Must be high to start a computation
pad_NoOutxSI	Output communication is skipped (For key/header data transmission)
pad_LastxSI	Input communication is skipped (For the Message Authentication Code (MAC) computation)
pad_InxDI[7:0]	Parallel data input (reused as ScanChain input)
pad_OutxDO[7:0]	Parallel data output (reused as ScanChain output)

A.8. Operation Modes

A.8.1. Functional Modes

- **pad_SelcUnit[1:0] = 0b00:** The 3-Share instance is active.
- **pad_SelcUnit[1:0] = 0b01:** The 3-Share-Star instance is active.
- **pad_SelcUnit[1:0] = 0b11:** The 4-Share instance is active.
- **pad_SelModexSI[0] = 0b0:** The chip is in encryption mode. During pad_NoOut is high, no output is transmitted to the user. Therefore, key and/or header data can be sent to the chip. Afterwards, for each 1088-bit block the chip sends back a 1088-bit block (cipher text). If the user raises pad_Last the chip ignores the input for the next transmission and outputs the MAC. After this transmission the encryption is done and the chip can be started for a new computation.
- **pad_SelModexSI[0] = 0b1:** The chip is in decryption mode. During pad_NoOut is high, no output is transmitted to the user. Therefore, key and/or header data can be sent to the chip. Afterwards, for each 1088-bit block the chip sends back a 1088-bit block (plain text). If the user raises pad_Last to high the chip ignores the input for the next transmission and outputs the MAC. After the transmission the encryption is done and the chip can be started for a new computation.
- **pad_SelModexSI[1] = 0b0:** The activated instance runs in NM.
- **pad_SelModexSI[1] = 0b1:** The activated instance runs in MM.
- **pad_SelNumberxSI[3:0]:** If this number is higher than zero the chip runs in HM. In Addition, this number defines the dummy operation quantity.

A.8.2. Test Modes

- **pad_ScanEnxTI:** If this signal is high, the chip runs in ScanChain-Test mode. A full scan chain approach is used to test the ASIC (not the Random-Access Memory (RAM) units). As in- and output pad_InxDI and pad_OutxDO can be used in parallel. In addition, pad_SelUnitxSI[1:0] = 0b11 and pad_SelModexSI[1:0] = 0b00 must be given!
- **pad_SelUnitxSI[1:0] = 0b11:** The chip runs in RAM-Testmode. To select between the different RAM macros on the chip SelMode is reused:
  - **pad_SelModexSI[1:0] = 0b01:** The RAM macro of the 3-Share instance can be tested. Therefore, the chip expects 608 bytes transmitted with the four way handshake protocol. The content of each byte is stored soaring from address `0x000` to `0x25F`. Afterwards, ZORRO sends the bytes back to the user, again with the help of the for way handshake protocol and soaring from address `0x000` to `0x25F`.

  - **pad_SelModexSI[1:0] = 0b10:** The RAM macro of the 3-Share-Star instance can be tested. Therefore, the chip expects 608 bytes transmitted with the four way handshake protocol. The content of each byte is stored soaring from address `0x000` to `0x25F`. Afterwards, ZORRO sends the bytes back to the user, again with the help of the for way handshake protocol and soaring from address `0x000` to `0x25F`.
  - **pad_SelModexSI[1:0] = 11:** The RAM macro of the 4-Share instance can be tested. Therefore, the chip expects 808 bytes transmitted with the four way handshake protocol. The content of each byte is stored soaring from address `0x000` to `0x327`. Afterwards, ZORRO sends the bytes back to the user, again with the help of the for way handshake protocol and soaring from address `0x000` to `0x327`.

A.9. Initialization

ZORRO can handle different modes of operations. The user is responsible to transmit the random values used by the ASIC. This section gives an overview of how many random values are needed for the initialization. All required numbers must be sent with the help of the four way handshake protocol **before** the computation starts.

- **3-Share instance**
  - **NM:** No random values are needed.
  - **HM:** 4 bytes are needed to initialize the Linear Feedback Shift Register (LFSR).
  - **MM:** 400 bytes are needed to initialize the three shares.
  - **SMM:** 404 bytes are needed. The first 4 bytes belong to the LFSR, the rest to the three shares.
- **3-Share-Star instance**
  - **NM:** No random values are needed.
  - **HM:** 4 bytes are needed to initialize the LFSR.
  - **MM:** 404 bytes are needed. The first 4 bytes belong to the LFSR, the rest to the states.
  - **SMM:** 404 bytes are needed. The first 4 bytes belong to the LFSR, the rest to the three shares.
- **4-Share instance**
  - **NM:** No random values are needed.
  - **HM:** 4 bytes are needed to initialize the LFSR.
  - **MM:** 600 bytes are needed to initialize the four shares.

– **SMM:** 604 bytes are needed. The first 4 bytes belong to the LFSR, the rest to the four shares.

A.10. Data transfer

Due to the fact that ZORRO was designed to minimize the area of each instance, it processes the data of a block in a non-linear way. This section should give an explanation of how the ASIC expects the input data.

All instances can only handle blocks of 1088 bits. The first lane, 64 bits, can be transmitted in a linear way. Afterwards, this is not any longer the case for the remaining bits. The following byte has to contain the next four bit of the first and second slice. The four most significant bits of the byte represent the bits of the first slice. The four least significant bits of the byte represent the bits of the second slice. This is done until all bits of the first and the second slice are transfered. The slice transfer is continued with slice number 3 and 4, 5 and 6, and so on. This is done until all slices are transfered to ZORRO. The transmission from ZORRO to the user is done in the same way. A pseudo code for the transmissions can be found in Algorithm 1.

Algorithm 1: A description of the data transfer to and from ZORRO.

```
input  : A block I of the size 1088
output: A block O of the size 1088
1  for $i \leftarrow 0$ to 135 do
2      if $i < 8$ then
3          InxDI = I[$(i \cdot 8) - 1 : (i - 1) \cdot 8$];
4      else
5          Offs = $((64 + 128 \cdot ((i - 8)/16))$;
6          InxDI[7:4] = I[Offs + $(((i - 7) \bmod 16) \cdot 4) - 1$ : Offs + $((i - 8) \bmod 16) \cdot 4$];
7          Offs = Offs + 64;
8          InxDI[3:0] = I[Offs + $(((i - 7) \bmod 16) \cdot 4) - 1$ : Offs + $((i - 8) \bmod 16) \cdot 4$];
9      end
10 end
11 Computation of Result;
12 for $i \leftarrow 0$ to 135 do
13     if $i < 8$ then
14         O[$(i \cdot 8) - 1 : (i - 1) \cdot 8$] = OutxDO;
15     else
16         Offs = $64 + 128 \cdot ((i - 8)/16))$;
17         O[Offs + $(((i - 7) \bmod 16) \cdot 4) - 1$ : Offs + $((i - 8) \bmod 16) \cdot 4$] =
           OutxDO[7:4];
18         Offs = Offs + 64;
19         O[Offs + $(((i - 7) \bmod 16) \cdot 4) - 1$ : Offs + $((i - 8) \bmod 16) \cdot 4$] =
           OutxDO[3:0];
20     end
21 end
```

Appendix B

Task Description

On the following pages, the original task description is given. It starts with a short introduction into the topic. Afterwards, the actual thesis description is given followed by the goals of the thesis. A short project plan is given next. Finally, the deliverables are noticed.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Semester Project at the Department of
Information Technology and Electrical Engineering

Autumn Semester 2013

Philipp Dunst

Low-Area ASIC Design of a Duplex-Based DPA-Resistant Authenticated Encryption System

September 9, 2013

Advisors:	Michael Muehlberghuber (IIS), ETZ J89, Tel. +41 44 632 57 45, mbgh@iis.ee.ethz.ch
	Michael Hutter (TU-Graz), Tel. +43 316 873 5541 Michael.Hutter@iaik.tugraz.at
Handout:	September 16, 2013
Due:	March 16, 2014

1 Introduction

In October 2012, the National Institute of Standards and Technology (NIST) announced [4] that Keccak [7, 8, 9, 10] has been selected as the winner of the SHA-3 hash competition [3]. The underlying building principle of Keccak is a so-called sponge construction [5] as illustrated in Figure 1.

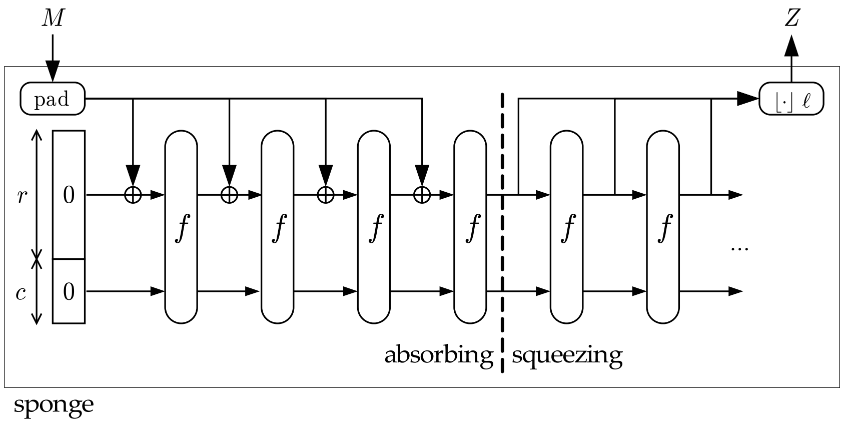


Figure 1: Sponge construction.[1]

Keccak is based on a new cryptographic hash family, the so-called sponge function family [5]. As opposed to existing hash algorithms, which are classically based on the Merkle-Damgård construction, a fixed length permutation f is used to allow the handling of arbitrary length inputs and to produce fixed length outputs, e.g., 224, 256, 384, or 512 bits. The permutations are performed on a state with a fixed size of b bits. The state is cut into two parts of size r (bitrate) and c (capacity), respectively. The bitrate defines the number of input bits, which are processed in one block permutation. The capacity of the sponge function represents the remaining bits of the state, i.e., $c = b - r$. The authors of Keccak proposed values for r and c in their submitted Keccak specification [10], e.g., $b = 1600$, $r = 1088$, and $c = 2n = 512$, where n is the length of the output.

Hashing works as follows. First, the state is initialized with zeros (0^b) and the input is padded to a length that is a multiple of r using the multi-rate padding scheme [8]. After that, it is cut into blocks of size r. During the initial *absorbing* phase, the message blocks are XORed with the first r bits of the state, followed by a single state permutation f. After the sponge has absorbed the whole message, it switches to the *squeezing* mode in which r bits are output iteratively (again followed by single state permutations if longer outputs are expected).

While Keccak represents the most famous example of cryptographic primitives based on such a two-phase sponge approach, sponge constructions and its derivative, the duplex constructions [6], are not limited to the use of hash functions. Authenticated encryption (AE) is one of the primitives, which can be realized based on such a duplex construction and assures both confidential and authentic transmissions between two communicating parties. As illustrated in

[1]The figure has been taken from `http://keccak.noekeon.org/` and is available under the Creative Commons Attribution license.

Figure 2, the core-component of a duplex-construction is also the underlying permutation f and its structure is rather similar to the one of a sponge-construction.

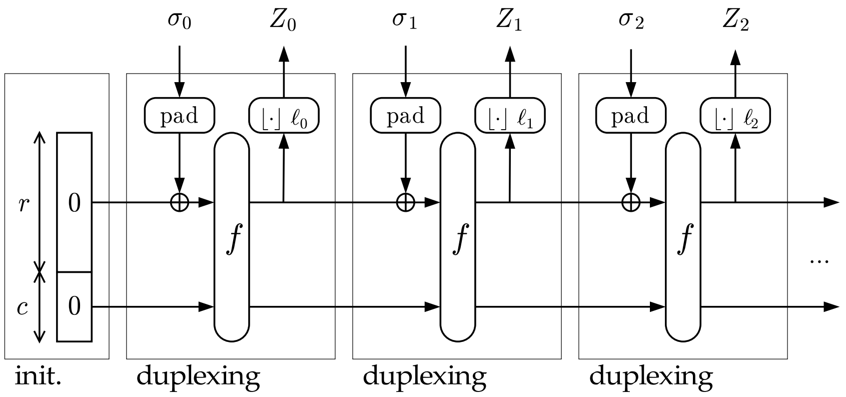


Figure 2: Duplex construction.[2]

Choice of the Permutation Function f

Since KECCAK has emerged as the winner of the SHA-3 hash competition back in 2012, several ideas on how to standardize KECCAK as the new SHA-3 have been proposed by NIST. Currently NIST prepares a FIPS draft standard including KECCAK with two capacities ($c = 256$ and $c = 512$), which will be available at the end of October 2013 according to John Kelsey from NIST [11]. SHA-3 will provide the same security level for pre-image resistance and collision resistance. Furthermore, the padding scheme will be different to the proposal of the authors. During this work we are going to focus on the permutation suggested by the KECCAK authors and adopted by NIST in order to realize the desired AE system.

2 Project Description

Throughout this project, we are focusing on the duplex-construction-based authenticated encryption (AE) service. The main goal of the work will be the development of a hardware architecture performing the overall AE process. Low area and low power will be of utmost importance for this design. Moreover, countermeasures against DPA attacks should be implemented.

2.1 Low-Area DPA-Secure ASIC Architecture

One of today's most powerful, side-channel-based attacking techniques are based on so-called differential power analysis (DPA) [12]. In order to protect our implementation from these attacks, Shamir's secret sharing technique [13] should be incorporated, i.e., DPA resistance

[2]The figure has been taken from `http://keccak.noekeon.org/` and is available under the Creative Commons Attribution license.

should be guaranteed through a masking technique as proposed by Bertoni et al. [15] and a hiding approach.

Masking. In this project, we achieve DPA resistance by applying a secret sharing technique. A variable is therefore represented by two or more shares where the sum (usually a bit-wise operation) equals to the native variable. The idea is to split all intermediate variables of the cryptographic implementation into several parts/shares. Thus, the native intermediate variable will not be processed by the device directly and can not be targeted by side-channel attacks. Only the shares of the variable will be processed which are generated randomly. Note that if $m+1$ shares are used, the countermeasure basically provides resistance to m-th order DPA/CPA (correlation power analysis) only (without considering glitch problems in dedicated hardware implementations, see Bertoni et al. [15]).

While secret sharing for linear operations is easy because the operation can be applied on each share independently, non-linear operations require special handling (non-trivial problem). In case of the permutation used within KECCAK, there is the only operation χ, which includes non-linear state permutations. All other four permutation functions are linear and can be easily combined. Thus the following requirements with regard to the masking countermeasure have to be fulfilled:

- For the linear operations only two shares should be used.
- For the non-linear operation three shares should be used.
- It should be possible to enable/disable the masking.

Hiding. In addition to secret sharing, a randomization technique (hiding [12]) should be included. The randomization should randomize/shuffle the permutations to hide the intermediate shares in addition to the masking countermeasure. This can be achieved by inserting dummy operations for different permutations. The higher the number of dummy operations, the higher the resistance against (first order) DPA attacks. Regarding the hiding technique, the following expectations should be met:

- The number of dummy operations should be configurable.
- It should be possible to enable/disable the hiding countermeasure.

2.1.1 Low-Area Design Techniques

The design of the AE scheme should target low-resource devices, so low power and area are the most important requirements. To achieve this goal, several optimizations might help (for more information on how to design a low-resource design of the KECCAK permutation see Pessl et al. [16]):

- Use of a RAM macro to store the (shared) state(s).
- Implement a Finite State Machine (FSM) for controlling.
- Round constants for ρ and ι should be stored in a look-up table (not in an LFSR).

- All shares should be processed sequentially (not in parallel).
- The round function has to be modified to separate all permutation functions.

2.1.2 Hardware Trojan

A hardware Trojan should be included in the design. The Trojan should disable the countermeasures individually (i.e., masking and hiding should be enabled/disabled separately). It consists of a comparator unit which compares the input for a certain message and should deactivate the countermeasure (set the masks to zero if enabled or apply no dummy permutations).

3 Goals

Throughout this work, the main goal is the development of a DPA-secure low-area ASIC architecture for authenticated encryption based on the duplex-construction. The results of the project should answer the following questions:

1. What is the area footprint of the authenticated encryption system based on the duplex-construction using the Keccak permutation?
2. How large is the area overhead for the implemented DPA countermeasures?
3. Is the DPA-protected version of the architecture still suitable for low-resource devices?
4. How large is the overhead of the masking compared to the hiding?
5. What is the overhead of the hardware Trojan?

4 Milestones

The following is a list of expected milestones of the project.

1. Project plan delivery
2. High-level model
3. Intermediate report delivery
5. VHDL implementation
6. Back-annotated ASIC netlist
7. Final report delivery

5 Project Realization

5.1 Project Plan

Within the first month of the project you will be asked to prepare a project plan. This plan should identify the tasks to be performed during the project and set deadlines for those tasks. The prepared plan will be a topic of discussion of the first week's meeting between you and your advisors. Note that the project plan should be updated constantly depending on the project's status. Figure 3 illustrates a preliminary project plan which may serve as a starting point for you. Furthermore, it already highlights some important events of your project.

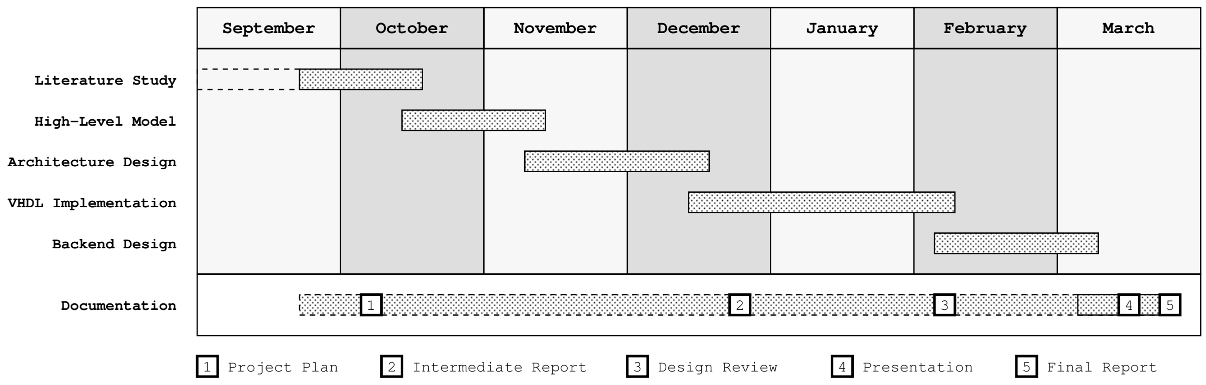


Figure 3: Preliminary project plan.

5.2 Meetings

Weekly meetings will be held between the student and the assistants. The exact time and location of these meetings will be determined within the first week of the project in order to fit the students and the assistants schedule. These meetings will be used to evaluate the status and progress of the project. Beside these regular meetings, additional meetings can be organized to address urgent issues as well.

5.3 VHDL Guidelines

In the following some guidelines for coding VHDL are given:

- **Naming Conventions:** Adapting a consistent naming scheme is one of the most important steps in order to make your code easy to understand. If signals, processes, and entities are always named the same way, any inconsistency can be detected easier. Moreover, if a design group shares the same naming convention, all members would immediately *feel at home* with each others code. At the IIS we make use of the naming convention proposed by the Microelectronics Design Zentrum [1]. Thus, try to maintain their naming convention in order to create readable and maintainable VHDL code.

- **Emacs Editor:** The preferred editor for writing VHDL code is the Emacs editor, as it has a really advanced VHDL mode. Because of this, you should get comfortable with the idea of using Emacs, even if you like a different editor.

5.4 Reports

Documentation is an important and often overlooked aspect of engineering. One short intermediate report and one final report (the Master Thesis) are to be completed within this study. Note that the intermediate report should be designed to be part of the final report.

The common language of engineering is de facto English. Therefore, the intermediate and final report of the work are preferred to be written in English. Any form of word processing software is allowed for writing the reports, nevertheless the use of LaTeX with Tgif[3] or any other vector drawing software (for block diagrams) is strongly encouraged by the IIS staff.

First Intermediate Report This report should be written in such a way to become the first part of your final report. It should contain general information about the topic, a description of the problem, explanations of related terminology, and descriptions of similar approaches in literature (with corresponding references to books, papers, etc.). It should roughly contain the "theoretical part" of your Master Thesis and has to be handed in before Christmas.

Final Report The final report has to be presented at the end of the Master Thesis and a digital copy as well as a printed one needs to be handed in and remains property of the IIS. This report is only accepted if the keys for the ETZ building as well as those for the student working room have been properly returned. Note that this task description is part of your thesis and has to be attached to your final report.

5.5 Design Review

Since the design developed throughout this project is supposed to be manufactured using an appropriate semi-conductor technology, a review of the chip design will be held during late November. The exact date of the review will be determined a few weeks in advance.

5.6 Presentation

There will be a presentation (20 min presentation and 5 min Q&A) at the end of this project in order to present your results to a wider audience. The exact date will be determined towards the end of the work.

[3]Tgif is a simple vector drawing software, quite useful for drawing block diagrams. For further information about Tgif we refer to `http://bourbon.usc.edu:8001/tgif/index.html` and `http://eda.ee.ethz.ch/index.php/Tgif`.

6 Deliverables

Throughout the project, the following deliverables have to be submitted in order to finish the work successfully:

- Project plan
- Intermediate report
- Final report
- Project CD/DVD incl. source codes, documentation, and presentation
- High-level model source files
- VHDL implementation
- Back-annotated netlist of final ASIC

References

[1] Design Zentrum website: `http://www.dz.ee.ethz.ch` and VHDL naming conventions: `http://eda.ee.ethz.ch/index.php/Naming_Conventions` (only available from within the ETHZ network)

[2] H. Kaeslin. "Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication". *Cambridge University Press*, 2008.

[3] National Institute of Standards and Technology (NIST). "SHA-3 Competition". `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`, [Online; accessed 03-September-2013].

[4] National Institute of Standards and Technology (NIST). "SHA-3 Selection Announcement". `http://csrc.nist.gov/groups/ST/hash/sha-3/sha-3_selection_announcement.pdf`, [Online; accessed 03-September-2013].

[5] G. Bertoni and J. Daemen and M. Peeters and G. Van Assche. "The Sponge Functions Corner". `http://sponge.noekeon.org/`, [Online; accessed 03-September-2013].

[6] G. Bertoni and J. Daemen and M. Peeters and G. Van Assche. "Cryptographic sponge functions". `http://sponge.noekeon.org/CSF-0.1.pdf`, [Online; accessed 03-September-2013].

[7] G. Bertoni and J. Daemen and M. Peeters and G. Van Assche. "The KECCAK sponge function family". `http://keccak.noekeon.org/`, [Online; accessed 03-September-2013].

[8] G. Bertoni and J. Daemen and M. Peeters and G. Van Assche. "The KECCAK reference". `http://keccak.noekeon.org/Keccak-reference-3.0.pdf`, Version 3.0, January 2011, [Online; accessed 03-September-2013].

[9] G. Bertoni and J. Daemen and M. Peeters and G. Van Assche. "KECCAK implementation overview". `http://keccak.noekeon.org/Keccak-implementation-3.2.pdf`, Version 3.2, May 2012, [Online; accessed 03-September-2013].

[10] G. Bertoni and J. Daemen and M. Peeters and G. Van Assche. "The Keccak SHA-3 submissionw". `http://keccak.noekeon.org/Keccak-submission-3.pdf`, Version 3, January 2011, [Online; accessed 03-September-2013].

[11] J. Kelsey. "SHA3 Past, Present, and Future". At *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2013, Invited Talk. `https://docs.google.com/file/d/0BzRYQSHuuMYOQXdHWkRiZXlURVE/edit`, [Online; accessed 03-September-2013].

[12] S. Mangard, E. Oswald, and T. Popp. "Power Analysis Attacks: Revealing the Secrets of Smart Cards". *Springer*, 2007.

[13] S. Nikova, V. Rijmen, M. Schläffer. "Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches ". In *Journal of Cryptology*, Vol. 24, 2011.

[14] G. Bertoni and J. Daemen and M. Peeters and G. Van Assche. "Building power analysis resistant implementations of Keccak". In *2nd SHA-3 Candidate Conference*, August, 2010.

[15] G. Bertoni, J. Daemen, N. Debande, T. Le, M. Peeters, and G. Van Assche. "Power Analysis of Hardware Implementations Protected with Secret Sharing". In *Cryptology ePrint Archive, Report 2013/067*, 2013.

[16] P. Pessl and M. Hutter. "Pushing the Limits of SHA-3 Hardware Implementations to Fit on RFID ". In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2013.

Zurich, September 9, 2013 Prof. Dr. Hubert Kaeslin

The thesis will not be accepted without returning the keys!

Bibliography

[1] AIST RCIS. SHA-3 Hardware Project. 2010. URL `http://www.rcis.aist.go.jp/special/SASEBO/SHA3-en.html`.

[2] M.-L. Akkar and C. Giraud. An implementation of DES and AES, secure against some attacks. In *Cryptographic Hardware and Embedded Systems—CHES 2001*, pages 309–318. Springer, 2001.

[3] B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O'Neill, and W. P. Marnane. FPGA Implementations of the Round Two SHA-3 Candidates. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 400–407. IEEE, 2010.

[4] G. Bertoni, J. Daemen, N. Veyrat-Charvillon, M. Peeters, and G. Van Assche. The KECCAK reference.

[5] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Building power analysis resistant implementations of Keccak. In *Second SHA-3 candidate conference*, 2010.

[6] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. KECCAK sponge function family main document. 2010.

[7] G. Bertoni, J. Daemen, P. Michael, and G. Van Assche. Cryptographic sponge functions. pages 1–93, 2011.

[8] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The KECCAK SHA-3 submission. 2011.

[9] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. pages 320–337, 2012.

[10] G. Bertoni, J. Daemen, N. Veyrat-Charvillon, M. Peeters, G. Van Assche, and R. Van Keer. Keccak implementation overview. 2012.

[11] B. Bilgin, J. Daemen, V. Nikov, S. Nikova, V. Rijmen, and G. Van Assche. Efficient and First-Order DPA Resistant Implementations of Keccak. 2013.

[12] P.-L. Cayrel, G. Hoffmann, and M. Schneider. GPU Implementation of the Keccak Hash Function Family. In *Information Security and Assurance*, pages 33–42. Springer, 2011.

[13] J. D. Golić and C. Tymen. Multiplicative masking and power analysis of AES. In *Cryptographic Hardware and Embedded Systems-CHES 2002*, pages 198–212. Springer, 2003.

[14] F. K. Gürkaynak, K. Gaj, B. Muheim, E. Homsirikamol, C. Keller, M. Rogawski, H. Kaeslin, and J.-P. Kaps. Lessons learned from designing a 65nm ASIC for evaluating third round SHA-3 candidates. In *Third SHA-3 Candidate Conference (March 2012)*, 2012.

[15] E. Homsirikamol, M. Rogawski, and K. Gaj. Comparing hardware performance of round 3 SHA-3 candidates using multiple hardware architectures in Xilinx and Altera FPGAs. pages 19–20, 2011.

[16] B. Jungk and J. Apfelbeck. Area-efficient FPGA implementations of the SHA-3 finalists. In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 235–241. IEEE, 2011.

[17] Kaeslin, Hubert. *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication.* Cambridge University Press, apr 2008.

[18] S. Kerckhof, F. Durvaux, N. Veyrat-Charvillon, F. Regazzoni, G. M. de Dormale, and F.-X. Standaert. Compact FPGA implementations of the five SHA-3 finalists. In *Smart Card Research and Advanced Applications*, pages 217–233. Springer, 2011.

[19] S. Mangard, T. Popp, and B. M. Gammel. Side-channel leakage of masked CMOS gates. In *Topics in Cryptology–CT-RSA 2005*, pages 351–365. Springer, 2005.

[20] S. Mangard, N. Pramstaller, and E. Oswald. Successfully attacking masked AES hardware implementations. In *Cryptographic Hardware and Embedded Systems–CHES 2005*, pages 157–171. Springer, 2005.

[21] S. Mangard, O. Elisabeth, and P. Thomas. *Power Analysis Attacks: Revealing the Secrets of Smart Cards.* Springer, 2007.

[22] A. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[23] R. Menicocci and J. Pascal. Elaborazione Crittografica di Dati Digitali Mascherati. Italian Patent IT MI0020031375A, 2003.

[24] A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. Pushing the limits: a very compact and a threshold implementation of AES. In *Advances in Cryptology–EUROCRYPT 2011*, pages 69–88. Springer, 2011.

[25] M. Muehlberghuber, C. Keller, F. K. Gürkaynak, and N. Felber. FPGA-Based High-Speed Authenticated Encryption System. pages 1–20, 2013.

[26] National Institute of Standards and Technology (NIST). SHA-3 Competition (2007-2012).

[27] National Institute of Standards and Technology (NIST). SHA-3 Selection Announcement. Nist, 2012. URL `http://csrc.nist.gov/groups/ST/hash/sha-3/sha-3_selection_announcement.pdf`.

[28] S. Nikova, C. Rechberger, and V. Rijmen. Threshold implementations against sidechannel attacks and glitches.

[29] S. Nikova, C. Rechberger, and V. Rijmen. Threshold implementations against side-channel attacks and glitches. In *Information and Communications Security*, pages 529–545. Springer, 2006.

[30] S. Nikova, V. Rijmen, and M. Schläffer. Secure hardware implementation of non-linear functions in the presence of glitches. In *Information Security and Cryptology–ICISC 2008*, pages 218–234. Springer, 2009.

[31] C. Paar and J. Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer, 2010.

[32] P. Pessl and M. Hutter. Pushing the Limits of SHA-3 Hardware Implementations to Fit on RFID. In *Cryptographic Hardware and Embedded Systems - CHES 2013*, pages 126–141. Springer Berlin Heidelberg, 2013.

[33] N. Pramstaller, E. Oswald, S. Mangard, F. K. Gürkaynak, and S. Häne. A Masked AES ASIC Implementation. In *Proceedings of Austrochip*, pages 77–82. Citeseer, 2004.

[34] P. Rogaway. Authenticated-encryption with associated-data. pages 98–107, 2002.

[35] A. Shamir. How to share a secret. volume 22, pages 612–613. ACM, 1979.

[36] J. Strömbergson. Implementation of the Keccak Hash Function in FPGA Devices. 2008.

[37] K. Tiri and I. Verbauwhede. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *Proceedings of the conference on Design, automation and test in Europe-Volume 1*, page 10246. IEEE Computer Society, 2004.

[38] E. Trichina. Combinational Logic Design for AES SubByte Transformation on Masked Data. volume 2003, page 236, 2003.

[39] E. Trichina, T. Korkishko, and K. H. Lee. Small size, low power, side channel-immune AES coprocessor: design and synthesis results. In *Advanced encryption standard–AES*, pages 113–127. Springer, 2005.

[40] E. Trichina, T. Korkishko, and K. H. Lee. Small size, low power, side channel-immune AES coprocessor: design and synthesis results. In *Advanced encryption standard–AES*, pages 113–127. Springer, 2005.

[41] Xilinx. Virtex-6 Family Overview, jan 2012. URL `http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf`.

[42] Xilinx. 7 Series FPGAs Configurable Logic Block, aug 2013. URL `http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf`.

[43] Xilinx. 7 Series DSP48E1 Slice, aug 2013. URL `http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf`.

[44] Xilinx. 7 Series FPGAs Overview, feb 2014. URL `http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf`.

[45] Xilinx. 7 Series FPGAs Memory Resources, jan 2014. URL `http://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf`.