Robert Schilling, BSc

# Securing the Communication- and Memory-Interfaces of a Multi-Core Cluster

**MASTER'S THESIS**

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

**Graz University of Technology**

Supervisor

Prof. Stefan Mangard

Institute of Applied Information Processing and Communications

Graz, March 2016

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

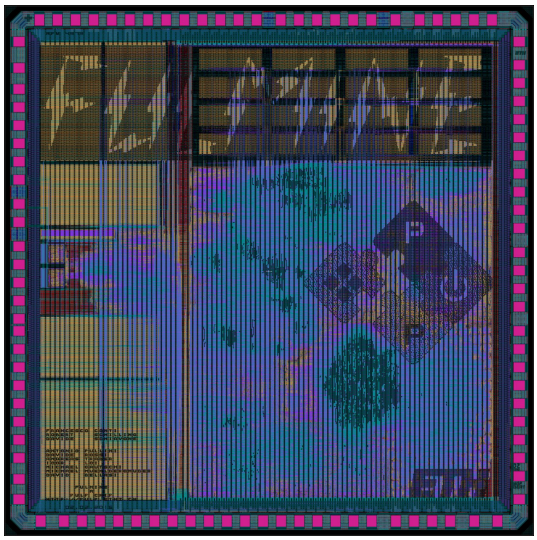
---

Department of Information Technology and
Electrical Engineering

Fall Semester 2015

# Securing the Communication- and Memory-Interfaces of a Multi-Core Cluster

Master Thesis



Robert Schilling
schrober@student.ethz.ch

March 2016

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

Professors: Prof. Dr. Luca Benini, lbenini@iis.ee.ethz.ch
Prof. Dr. Stefan Mangard, stefan.mangard@iaik.tugraz.at

# Acknowledgements

First and foremost, I want to thank Frank Gürkaynak and Michael Mühlberghuber from the Integrated Systems Laboratory at ETH Zurich for their help and support during this project. Thank you for all the fruitful discussions and always asking the right questions. I also want to thank Stefan Mangard from Graz University of Technology. Without his help this thesis would not be possible. Moreover, I want to thank all members of the PULP team of the ETH Zurich and University of Bologna for their support with the complex PULP system. Also thank you to the team of the Institute for Applied Information Processing and Communications of Graz University of Technology for all the valuable input on this project.

Finally, I must express my gratitude to my parents and to my girlfriend Anna for providing continuous support throughout my studies and this master's thesis. This time would not have been possible without them. Thank you for everything.

# Abstract

The Internet of Things (IoT) is a growing market. More and more different kinds of devices are being connected and share sensible information via their communication interfaces. In order to ensure that only the intended receiver can read the transmitted data, IoT devices use encryption algorithms to protect the communication. Since those devices are deployed in the field, adversaries have physical access to them and can perform side-channel attacks on the cryptographic algorithms. To counteract side-channel attacks, such as the differential power analysis, we evaluate different leakage-resilient encryption algorithms in a hardware implementation. In this thesis, we develop a cryptographic hardware accelerator, which should protect a multi-core processor system against state-of-the-art side-channel attacks. Counteracting attacks like DPA is accomplished based on the concept of fresh re-keying, which ensures that an encryption key is only used for one single encryption. We implement two different encryption modes which rely on this principle. First, we design a re-keying function based on a polynomial multiplication in combination with a leakage-resilient bulk encryption mode using the Advanced Encryption Standard. Second, we present the first hardware implementation of PASEC, a leakage-resilient authenticated encryption framework build on a permutation function. PASEC supports two re-keying functions, one encryption and one authentication mode. All re-keying functions and encryption modes can be combined, which results in a highly-flexible cryptographic accelerator. The accelerator is integrated into a multi-core processing architecture to fulfill two properties. First, the internal memory can be encrypted. Adversaries are not able to read the encrypted data in the memory. Second, a secure communication via the peripheral interfaces is possible. Both scenarios are protected against side-channel attacks which are reasonable in a hostile environment. The cryptographic accelerator is taped-out as part of the *Fulmine* application specific integrated circuit (ASIC) using the 65 nm technology of United Microelectronics Corporation. *Fulmine* represents the first ASIC implementation of the presented cryptographic modes available to date.

**Keywords:** Internet of Things, leakage-resilient cryptography, fresh re-keying, side-channel attacks, differential power analysis, application specific integrated circuit

# Kurzfassung

Das Internet der Dinge ist ein wachsender Markt. Immer mehr verschiedene Geräte werden vernetzt und kommunizieren miteinander. Zum Schutz der Kommunikation werden Verschlüsselungsalgorithmen eingesetzt, die sicherstellen, dass nur der beabsichtigte Empfänger die Nachricht lesen kann. Da Geräte des Internets der Dinge an den unterschiedlichsten Orten eingesetzt werden, haben potentielle Angreifer physikalischen Zugriff darauf. Dies ermöglicht den Angreifern die Durchführung von Seitenkanalangriffen auf den Geräten. In dieser Diplomarbeit werden verschiedene Verschlüsselungsvarianten evaluiert, die speziell gegen Seitenkanalangriffe wie eine differentielle Leistungsanalyse abgesichert sind. Es wird ein kryptographischer Hardware Beschleuniger entwickelt, der einen Mehrkern-Prozessor gegen modernste Seitenkanalangriffe absichert. Das Konzept der frischen Schlüsselgenerierung kommt zum Einsatz um eine differentielle Leistungsanalyse zu verhindern. Dabei wird gewährleistet, dass ein Schlüssel nur für exakt eine Verschlüsselung verwendet wird. In dieser Arbeit werden zwei Algorithmen in Hardware implementiert, die auf diesem Prinzip basieren. Der erste Modus beschreibt eine Schlüsselfunktion basierend auf einer Polynommultiplikation. Diese Funktion wird mit einem Verschlüsselungsmodus kombiniert, der auf einem Advanced Encryption Standard beruht. Als zweiten Modus präsentiert diese Arbeit die erste Hardware Implementierung von PASEC, einem authentifizierten Verschlüsselungsmodus basierend auf einer Permutation, der zwei Schlüsselfunktionen, einen Verschlüsselungsmodus sowie einen Authentifizierungsmodus, unterstützt. Alle Schlüsselfunktionen und Verschlüsselungsmodi sind miteinander kombinierbar, wodurch sich eine Vielzahl von verschiedenen Konfigurationen für den flexiblen kryptographischen Beschleuniger ergibt, um zwei Aufgaben zu erfüllen. Zum einen kann der interne Speicher verschlüsselt werden, weshalb es Angreifern nicht mehr möglich ist, die Daten im Speicher zu verstehen. Des Weiteren kann mit einem kryptographischen Beschleuniger eine sichere Kommunikation über Peripherieschnittstellen gewährleistet werden, die zudem gegen Seitenkanalangriffe abgesichert ist. Der kryptographische Beschleuniger wird als Bestandteil des *Fulmine* Chips in der 65 nm Technologie von United Microelectronics Corporation produziert. *Fulmine* ist die erste Hardware Implementierung der präsentierten Verschlüsselungsvarianten, die derzeit verfügbar ist.

**Stichwörter:** Internet der Dinge, Seitenkanalangriffe, Differentielle Leistungsanalyse, Integrierte Schaltungen

## AFFIDAVIT

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 indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

14.03.2016
Date

Signature

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Acronyms

AES . . . . . Advanced Encryption Standard
ASIC . . . . . Application-Specific Integrated Circuit
AXI . . . . . Advanced Extensible Interface

CPA . . . . . Correlation Power Analysis
CPU . . . . . Central Processing Unit
CRT . . . . . Chinese Remainder Theorem

DES . . . . . Data Encryption Standard
DFA . . . . . Differential Fault Analysis
DFT . . . . . Design for Testability
DMA . . . . . Direct Memory Access
DPA . . . . . Differential Power Analysis
DUV . . . . . Design under Verification

ECB . . . . . Electronic Codebook
ECDH . . . . . Elliptic-Curve Diffie-Hellman

FPGA . . . . . Field Programmable Gate Array
FSM . . . . . Finite State Machine

GCM . . . . . Galois Counter Mode

GE . . . . . .Gate Equivalent

GF . . . . . .Galois Field

HDL . . . . . .Hardware Description Language

$I^2C$ . . . . . .Inter-Integrated Circuit

IAIK . . . . . .Institute for Applied Information Processing and Communications

IC . . . . . .Integrated Circuit

IIS . . . . . .Integrated Systems Laboratory

IoT . . . . . .Internet of Things

IRQ . . . . . .Interrupt Request

LFSR . . . . . .Linear Feedback Shift Register

LR . . . . . .Leakage-Resilient

LUT . . . . . .Lookup Table

MAC . . . . . .Message Authentication Code

NIST . . . . . .National Institute of Standards and Technology

PRF . . . . . .Pseudo Random Function

PRG . . . . . .Pseudo Random Generator

PRNG . . . . . .Pseudo Random Number Generator

PULP . . . . . .Parallel Ultra-Low-Power Processing-Platform

R2R . . . . . .Register to Register

RAM . . . . . .Random Access Memory

RFID . . . . . .Radio-Frequency Identification

RFU . . . . . .Reserved for future use

RTL . . . . . .Register Transfer Level

SNR . . . . . .Signal to Noise Ratio

SoC . . . . . .System-on-Chip

SPA . . . . . Simple Power Analysis

SPI . . . . . Serial Peripheral Interface

TCDM . . . . . Tightly Coupled Data Memory

UART . . . . . Universal Asynchronous Receiver Transmitter

UMC . . . . . United Microelectronics Corporation

UVM . . . . . Universal Verification Methodology

XEX . . . . . XOR-Encrypt-XOR

XTS . . . . . XEX-based tweaked-codebook mode with ciphertext stealing

Chapter 1

# Introduction

The growing market of the Internet of Things (IoT) started about ten years ago when the first devices became *smart*. With the last IPv4 addresses handed out in 2011, the market of connected devices is steadily growing. New devices in the domain of IoT are not traditional computers or smartphones anymore. Instead, in the last few years, a lot of different types of devices got an interface to a network. These new computing devices are the Internet of Things. In fact, IoT devices can be anything. Refrigerators, sensor-nodes, lamps, or even cars are connected. Outlooks to the year 2020 predict about 50 billion running IoT devices. Due to the connecting interfaces, end-users can control these devices in a *smart* way, which is done using applications on the smartphone or via the computer. Although this evolution of technology is very impressive, the development of IoT devices is still in its early days. These devices already have a comprehensive feature set. However, this also increases the risk of being hacked.

IoT devices operate in a hostile environment. Since such devices are deployed in the field, almost everyone has access to them. This allows adversaries to mount physical attacks on them. In the last year, two security researchers showed that the infotainment center of a Jeep Cherokee car had major vulnerabilities [2]. The security researchers were able to exploit them to take over a running car, and had control over the brakes and the transmission unit. This example shows that hackers could induce a car crash via a computer on the other side of the world. This motivates security researchers to develop new mechanisms for secure communications to avoid a scenario like that.

In 2009 Halderman et al. [3] presented a novel attack on dynamic random access memories (DRAM). He and his team showed that this type of memory does not loose the stored data immediately when they are turned off. Instead, deleting the content happens gradually over time. When keeping the temperature of the memory low, the deleting process even slows down. It was shown that the data was stored for hours in a turned-off memory. Attackers can remove the memory from a computer and use a special test setup

to readout the memory content to gain information of sensible data stored in the DRAM. This is called *cold boot* attack.

A physical side-channel attack on ordinary laptops is published by Genkin et al. [4]. They gain information out of the electromagnetic leakage of the computer to attack the elliptic-curve Diffie-Hellman (ECDH) algorithm to extract the private key. This works even if the measurement setup is in a room next to the attacked computer. They attack the *libgcrypt* library, which is used by various applications. Contrary to other attacks, this particular attack recovers the key within seconds.

These vulnerabilities are only a few examples, but they show one thing. The implementation of security protocols and their algorithms are still in their early stages of development concerning side-channel resistance. It is always a cat and mouse game between people who develop products and people who attack these devices. To improve the security of pervasive devices, it requires the industry and the research community to develop and analyze proper solutions.

All these devices are based on a central processing unit and use cryptographic algorithms to protect the memory and the communication. In this thesis, we first consider the memory encryption scenario. The internal and external memory is encrypted in order that an adversary does not gain any information out of it. Second, we consider the secure communication scenario. An IoT device has different interfaces for communication. All data sent over such communication interfaces shall be encrypted. Thereby, attackers are not able to read the original data anymore. Both scenarios should include countermeasures to protect the implementation against side-channel attacks.

## 1.1 Contribution of this Work

In this thesis, we develop a highly-performance cryptographic accelerator for the PULP System-on-Chip (SoC). This accelerator implements leakage-resilient encryption modes to withstand sophisticated side-channel attacks. Thereby, we implement state-of-the-art cryptographic algorithms in hardware for a highly-flexible evaluation platform. The architecture uses the concept of fresh re-keying to implement an AES-128 based leakage-resilient stream-cipher which facilitates a fast key-update function based on a pseudo random generator. Furthermore, We present the first hardware implementation of the permutation-based authenticated encryption framework PASEC. Additionally, we implement AES-128 ECB and AES-128 XTS encryption for comparison purposes. Finally, the accelerator supports the access of the internal primitives to implement hardware-accelerated algorithms in software. The cryptographic accelerator is integrated into the PULP system and taped-out as part of the *Fulmine* application-specific integrated circuit (ASIC) using the United Microelectronics Corporation (UMC) 65 nm technology. *Fulmine* represents the first multi-core microprocessor containing high-performance state-of-the-art leakage-resilient encryption approaches available to date.

## 1.2 Organization

The thesis is organized as follows. In Chapter 2 we give an introduction to cryptography. Furthermore, we describe side-channel attacks and provide an understanding to countermeasures for this kind of attacks. Chapter 3 presents existing solutions and implementations of leakage-resilient cryptography and cryptographic extensions to commercial products. In Chapter 4 we show the theoretical background of this thesis. We present the implemented algorithms and modes of operation in detail. Chapter 5 describes the hardware architecture of the cryptographic accelerator *HWCrypt*. Furthermore, we describe the concrete design and the design decisions. Moreover, we present the functional verification flow, which is used to test the implementation using a pure RTL approach, as well as a hardware-software co-simulation. Eventually, we discuss in Chapter 6 performance results of software and hardware implementations of the algorithms used in this thesis. Finally, we draw in Chapter 7 a conclusion of this work and give an outlook for the future.

Chapter 2

# Background

In this chapter, we introduce the PULP multi-core processing architecture. In the following section we give an introduction to cryptography. We show the basic principles and primitives. Furthermore, we describe the Advanced Encryption Standard (AES) and the KECCAK-$f$ permutation family, which are used in this work. Eventually, we give an introduction to side-channel attacks and their countermeasures.

## 2.1 PULP - A Parallel Ultra-Low Power Processing Architecture

PULP is a heterogeneous multi-core System-on-Chip (SoC) platform for the context of IoT applications. The goal of this processing architecture is to design a processor, which fulfills the demands of state-of-the-art IoT applications but with a highly-optimized energy consumption. PULP is developed by the ETH Zurich in collaboration with the University of Bologna.

Figure 2.1 shows the architecture of a PULP chip. The main part of this architecture is the cluster, which contains multiple processing cores depicted with the green boxes. The processing core can either be an OpenRISC [5], or a RISC-V [6] core and have access to a level-1 memory, which is called the tightly coupled data memory (TCDM). Access to this memory is fast and can be achieved in one clock cycle in the best case. This memory is used in software like a scratchpad. Data can be loaded into the cluster either via direct memory accesses (DMA) or directly by the processing cores using the memory bus. Outside the cluster the SoC contains different peripheral controllers such as a Universal Asynchronous Receiver Transmitter (UART), a Serial Peripheral Interface (SPI), or an Inter-Integrated Circuit ($I^2C$) bus. Furthermore, the SoC contains a level-2 (L2) memory which contains the executable and user data.

For energy efficiency reasons most of the parts can be clock-gated to reduce their dynamic power consumption. In addition to that, the processing cores support a sleep mode, in which the cores itself are clock-gated and wait for an event.

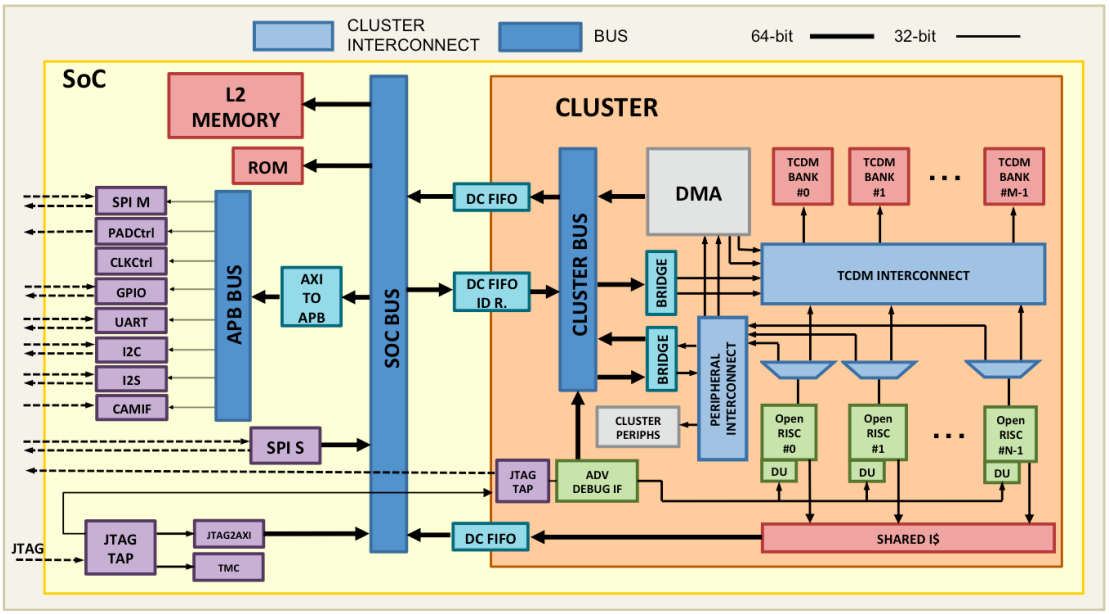



Figure 2.1: PULP architecture.

## 2.2 Introduction to Cryptography

This section aims to give an overview about the basic principles in cryptography, which later chapters are based on. Cryptography has a long history. Even in the ancient Rome, people used ciphers to encrypt important messages during their transport. This lead to the famous Caesar-cipher, a substitution cipher which replaces every letter of a message. Thereby, it uses a fixed shift in the alphabet to determine the new encrypted character. Since then, cryptography evolved a lot. Today, modern cryptography tries to achieve the following four properties:

**Confidentiality.** This property protects transmitted data between two entities, such that unauthorized third-parties are unable to read the transmitted data. An eavesdropper only can read the ciphertext but does not understand it.

**Integrity.** With integrity the transmitted message is protected against altering. Integrity proves that the transmitted message is correct. The receiving party is able to detect an altered message and can therefore discard it.

**Authentication.** With authentication, we can ensure that a message is guaranteed to come from the claimed entity. If the entity Bob is receiving a message with a message authentication code (MAC) from Alice, the MAC ensures the message is not coming from another entity.

**Non-Repudiation.** This concept ensures that an entity cannot deny a previous made transfer of a message.

Researchers developed two leading cryptographic schemes to implement the properties stated above. This leads to the concepts of asymmetric and symmetric cryptography.

### 2.2.1 Asymmetric Cryptography

Asymmetric cryptography uses two types of keys for different purpose, namely the public and the private key. The public key, as its name suggests, is public and available for everyone. This key is used to encrypt a message. The second key is the private key, which is kept secret. Only when the corresponding private key is known, one is able to decrypt a message. This can be simplified, to a padlock and a corresponding key to open it, as depicted in Figure 2.2. The public key equals the padlock. Everyone can get the padlock to enclose a message. Only the entity with the key is able to open the padlock again and read the enclosed message. There are two wide-spread algorithms in cryptography, which rely on this principle, namely the RSA scheme and elliptic-curve cryptography.

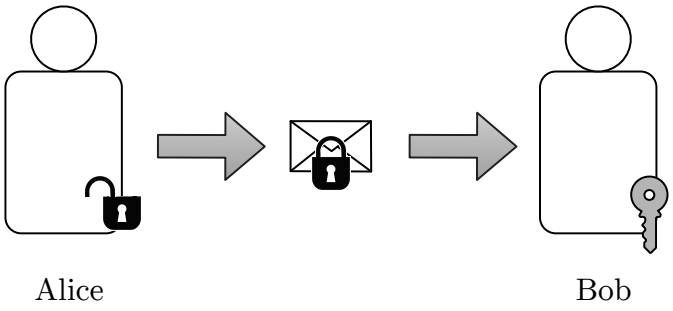



Figure 2.2: Asymmetric cryptography scheme.

### 2.2.2 Symmetric Cryptography

Symmetric cryptography uses a shared secret to implement a confidential communication as indicated in Figure 2.3. As the name suggests, both entities Alice and Bob use the same shared secret, namely the key $K$. With this key Alice is able to encrypt a message $M$ to the ciphertext $C$ and also to decrypt it. An eavesdropper is computationally not able to recover the original message $M$ from the ciphertext. The receiver Bob uses the same

secret key $K$, which was used to encrypt the message, to recover the message $M$ again. However, such a scheme requires both parties to know the shared key $K$ in advance. Therefore, key exchange protocols, which may are based on asymmetric cryptography, are used to transfer the shared key.

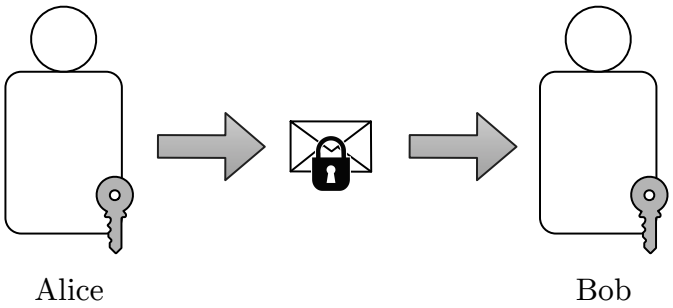



Figure 2.3: Symmetric cryptography scheme.

As stated in Section 2.2, cryptography aims to solve the problems of confidentiality, integrity, and authenticity. In the past, dedicated algorithms were used to achieve each of these requirements separately. Since each algorithm needs to be invoked, this lead to overhead in runtime, required code, or required area when talking about hardware implementations. To improve the performance, developers developed the concept of *authenticated encryption*, which aims to combine the individual steps into one algorithm.

To implement symmetric cryptographic primitives, either a block cipher or a stream cipher may be used as described in the following section.

### 2.2.3 Block Cipher

A block cipher encrypts a plaintext block $P = \{0,1\}^m$ with the fixed block size $m$ into a ciphertext $C$ of the same size. Every bit of the ciphertext depends here on every bit of the plaintext. Typical block sizes for $m$ are 64-bit or 128-bit. Figure 2.4 shows a block cipher, which encrypts each block of data independently using the Electronic Codebook (ECB) mode. The block cipher is able to encrypt and decrypt the input data denoted by `E/D`. In this architecture the plaintext $P$ is encrypted to the ciphertext $C$ using the key $K$. If the block cipher only supports the encryption mode, this is denoted by `E`.

### 2.2.4 Stream Cipher

A stream cipher encrypts every bit of the plaintext $P$ individually by XORing it with an encryption pad. The encryption pads, also called the pad-stream, are computed via a pad generator. This pad generator, which depends on the secret key $K$, computes a pseudo-random sequence of bits. In synchronous stream ciphers the pad-stream only

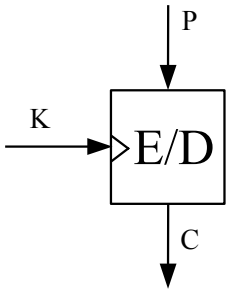



Figure 2.4: Block cipher in the ECB mode.

depends on the secret key $K$. However, in asynchronous stream ciphers, the pad-stream is computed based on both the key $K$ and the previously encrypted ciphertext.

## 2.3 Advanced Encryption Standard

The Advanced Encryption Standard (AES) [7] is a block cipher and the successor of the Data Encryption Standard (DES). The AES algorithm *Rijndael* is the winner of a competition to find a new encryption algorithm to avoid the weaknesses of DES. AES supports a block-size of 128-bit and a key-size of 128-, 192-, and 256-bit, referred to as AES-128, AES-192, and AES-256. These algorithms differ in the number of invoked round functions and their key expansion algorithm. For the sake of simplicity, we now describe the AES-128 algorithm.

AES organizes the 128-bit state $S$ in a 4x4 matrix as shown in Figure 2.5. Each value $S_{r,c}$ in this matrix represents one element in the finite field GF($2^8$) using the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. Given the 128-bit (16-byte) input block $I$, the state is initialized using the relation $S_{r,c} = I[r + 4c]$. AES uses four core operations to construct an encryption round. Those core operations are described below.

| | | | |
|---|---|---|---|
| $s_{0,0}$ | $s_{0,1}$ | $s_{0,2}$ | $s_{0,3}$ |
| $s_{1,0}$ | $s_{1,1}$ | $s_{1,2}$ | $s_{1,3}$ |
| $s_{2,0}$ | $s_{2,1}$ | $s_{2,2}$ | $s_{2,3}$ |
| $s_{3,0}$ | $s_{3,1}$ | $s_{3,2}$ | $s_{3,3}$ |

Figure 2.5: AES state matrix $S$.

**AddRoundKey.** In the *AddRoundKey* operation a round-key is added to the current state using the XOR operation. Round-keys are computed by executing the key expansion algorithm as defined in Algorithm 3.

**SubBytes.** The *SubBytes* operation, as depicted in Figure 2.6, transforms all bytes of the state by using an invertible S-box. This substitution is the only non-linear operation in the cipher. The S-box is based on the multiplicative inverse in the finite field GF($2^8$) followed by an affine transformation.

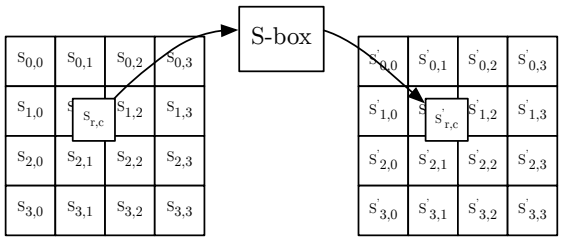



Figure 2.6: AES *SubBytes* operation.

**ShiftRows.** The *ShiftRows* operation in AES only operates on the rows of the state $S$. As indicated in Figure 2.7, the first row is left untouched. Beginning with the second row, the row is cyclically shifted by one to the left. The third row is shifted by two bytes to the left. Eventually, the fourth row is shifted by three bytes to the left. This operation provides a good diffusion over the columns.

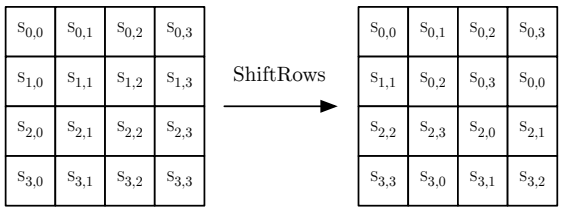



Figure 2.7: AES *ShiftRows* operation.

**MixColumns.** The *MixColumns* step operates on each column independently. Each column is treated as a polynomial and is multiplied with a fixed polynomial. This is indicated with the matrix multiplication shown in Figure 2.8. This operation provides a good intra-column diffusion.

AES uses these four operation to construct one round, which is iteratively applied on the state $S$. Algorithm 1 describes the encryption algorithm of AES-128. The AES operations are applied ten times (ten rounds), but during the last round the *MixColumn* step is left out. Each round uses a different round-key during the *AddRoundKey* operation, which is computed by the key expansion algorithm.

The decryption algorithm, as shown in Algorithm 2, operates in the reversed order compared with the encryption algorithm. Moreover, it uses the inverse operation of the AES operations defined in Section 2.3. Since the round-keys are the same, but in a reverse

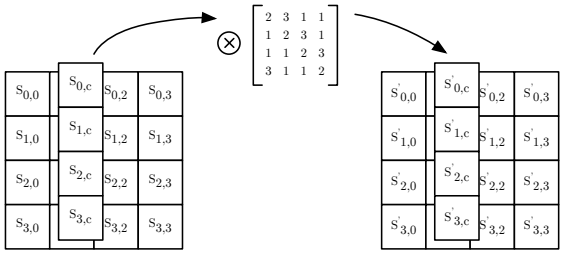



Figure 2.8: AES *MixColumn* operation.

**Algorithm 1:** AES-128 encryption function.

**Input**: $P \in [0, 2^{128} - 1], RK_{0,..,10} \in [0, 2^{128} - 1]$
**Output**: $C \in [0, 2^{128} - 1]$

```
S ← AddRoundKey(P, RK_0);
for i ← 1 to 10 do
    S ← SubBytes(S);
    S ← ShiftRows(S);
    if i < 10 then
        S ← MixColumns(S);
    end
    S ← AddRoundKey(S, RK_i);
end
return S
```

order, the key expansion algorithm stays the same for decryption. Each of the AES operations described above supports an inverse operation including the non-linear *SubBytes* transformation. The *AddRoundKey* operation is the same for both encryption and decryption.

The key expansion algorithm of AES-128 expands the initial 128-bit encryption key $K$ into 11 round-keys $RK_0$ to $RK_{10}$ as described in Algorithm 3. This algorithm iteratively computes the next round-key based on the previous one. It operates on words, each 32-bit wide. The *RotWord* operation cyclically shifts one 32-bit word by one byte. The *SubWords* operation substitutes all four bytes by applying the same S-box lookup as used during encryption. The key expansion algorithm returns an array of 44 words containing the eleven 128-bit round-keys $RK_i$. The first 128-bit round-key equals the encryption key $K$. As indicated by Algorithm 3, the key expansion algorithm requires the round constants $RCon_i$, which are defined in [7].

**Algorithm 2:** AES-128 decryption function.

**Input**: $C \in [0, 2^{128} - 1], RK_{0,..,10} \in [0, 2^{128} - 1]$
**Output**: $P \in [0, 2^{128} - 1]$

$S \leftarrow \text{AddRoundKey}(P, RK_{10})$;
**for** $i \leftarrow 9$ **to** $0$ **do**
  $S \leftarrow \text{InvShiftRows}(S)$;
  $S \leftarrow \text{InSubBytes}(S)$;
  **if** $i > 0$ **then**
    $S \leftarrow \text{InvMixColumns}(S)$;
  **end**
  $S \leftarrow \text{AddRoundKey}(S, RK_i)$;
**end**
**return** $S$

**Algorithm 3:** AES-128 key expansion function.

**Input**: $K \in [0, 2^{128} - 1]$
**Output**: $RK_{0..43} \in [0, 2^{32} - 1]$

**for** $i \leftarrow 0$ **to** $3$ **do**
  $RK_i \leftarrow K_i$;
**end**
**for** $i \leftarrow 4$ **to** $43$ **do**
  $tmp \leftarrow RK_{i-1}$;
  **if** $i \bmod 4 = 0$ **then**
    $tmp \leftarrow SubWord(RotWord(tmp)) \oplus RCon_{i/4}$;
  **end**
  $RK_i \leftarrow K_{i-4} \oplus tmp$;
**end**
**return** $RK_{0..43}$

## 2.4 Sponge Construction

The sponge construction [8] is a flexible cryptographic design to implement various cryptographic primitives, such as hash functions, encryption modes, or authentication functions. The heart of a sponge construction is the permutation function $p$, which is iteratively applied on a state $S$. The state $S$ is split into a rate-part $r$ and a capacity-part $c$. The rate-part is used to operate with external data. The capacity-part is used as an internal state, which is not visible to the outside. Figure 2.9 shows a classic sponge construction, which is used to construct a hash function.

A sponge construction supports two different operation phases. The first operation is the *absorbing* phase. The messages $m_i$ are absorbed into the state, followed by an application

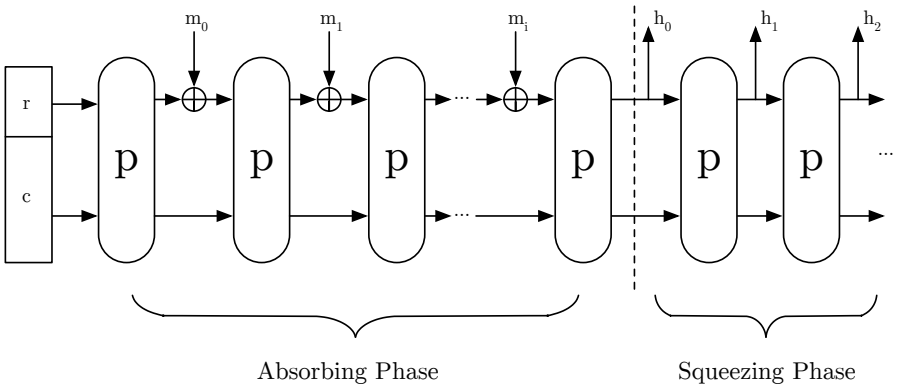



Figure 2.9: Sponge construction for a hash function.

of the permutation function $p$. This is repeated for any arbitrary length of data. The second operation is the *squeezing* phase. In this mode, the rate-part $r$ of the state is used as the output. In the hash construction of Figure 2.9 this equals the hash of the message.

### 2.4.1 The Keccak-$f$ Permutation Family

The SHA-3 competition, which aimed to find a new standard for hash functions, was won by the Keccak [9] algorithm. The Keccak hash algorithm is a sponge construction which uses a permutation as its base operation. This permutation, or better, the family of these permutation functions are described here.

The permutation family Keccak-$f$ describes seven different permutation functions, which only differ in their bit-width denoted by $b \in 25, 50, 100, 200, 400, 800, 1600$ and the number of rounds $n_r$. The number of rounds $n_r$ is given by $n_r = 12 + 2l$, where $2^l = b/25$. For Keccak-$f$[25] this returns 12 rounds, and for Keccak-$f$[1600] this results in 24 rounds respectively. The Keccak-$f$ permutation function iteratively applies different functions on the state $S$. Keccak organizes the state as a 5x5x$2^l$ cube-matrix as illustrated in Figure 2.10. The z-axis grows depending on the configuration parameter $l$.

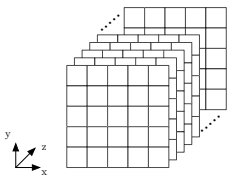



Figure 2.10: Keccak state matrix $S$.

The state is organized in lanes, slices, rows, and columns as shown in Figure 2.11. The

core functions of KECCAK, on which the round function is based on, use these representations for their operation. KECCAK uses the five core functions $\theta$, $\rho$, $\pi$, $\chi$, and $\iota$ to construct one round. This round is invoked multiple times to construct the KECCAK permutation, which is described in Algorithm 4. The functionality description of the core functions below only gives a rough overview rather than a fully detailed description.

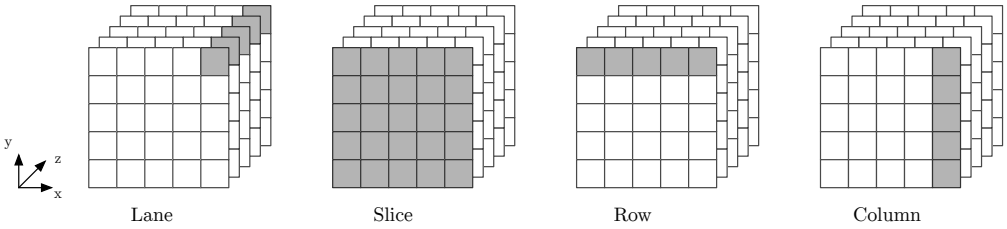



Figure 2.11: KECCAK state organization.

**Description of the $\theta$ Function.** This step adds to each bit $S[x][y][z]$ the bitwise sum of the columns $S[x-1][\cdot][z]$ and $S[x+1][\cdot][z-1]$.

$$S[x][y][z] \leftarrow S[x][y][z] + \sum_{i=0}^{4} S[x-1][i][z] + \sum_{i=0}^{4} S[x+1][i][z-1] \tag{2.1}$$

**Description of the $\rho$ Function.** The $\rho$ operation performs a cyclic rotation with a constant shift $C$ within each lane of the state. The shift constants $C_{x,y}$ are defined in [9].

$$S[x][y][z] \leftarrow S[x][y][z] + C_{x,y} \tag{2.2}$$

**Description of the $\pi$ Function.** This operations permutes the lanes as indicated in Equation 2.3 and Equation 2.4. Since this step operates on each slice independently, it is invariant to the z-axis. Equation 2.4 shows the assignment of the matrix multiplication of Equation 2.3 .

$$\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.3}$$

$$S[x][y][z] \leftarrow S[y][2 \cdot x + 3 \cdot y][z] \tag{2.4}$$

**Description of the $\chi$ Function.** The $\chi$ operation is the only non-linear operation in this construction. It applies a non-linear mapping on the 5-bit rows, which acts as an S-box lookup.

$$S[x][y][z] \leftarrow S[x][y][z] \oplus ((\neg S[x+1][y][z]) \,\&\, S[x+2][y][z]) \tag{2.5}$$

**Description of the $\iota$ Function.** This step adds round-dependent constants to the state. These constants are defined in [9]. Without this operation the permutation function would be symmetric.

Algorithm 4 shows the iterative usage of the base functions defined above. In each round, a round-dependent constant $RCon_i$ is added to the state during the $\iota$ operation.

**Algorithm 4:** KECCAK-$f$[b] permutation function.

**Input**: $S \in [0, 2^b - 1], b \in 25, 50, 100, 200, 400, 800, 1600, RCon_{0..n_1-1}$
**Output**: $S \in [0, 2^b - 1]$

```
for i ← 0 to n_r − 1 do
    S ← θ(S));
    S ← ρ(S));
    S ← π(S);
    S ← χ(S);
    S ← ι(S, RCon_i);
end
return S
```

## 2.5 Side-Channel Attacks

In a traditional cryptographic system a primitive operates on certain input data. As depicted in Figure 2.12, the cryptographic system uses an internal secret to process the input data and compute the output data. Encryption of data is one example for such a system. In a perfect world, such an architecture would be safe, as long as the underlying cryptographic primitive is mathematically secure.

However, life is not perfect. In reality a system as described before leaks information about the internal computation via so called side channels. This phenomenon is illustrated in Figure 2.13. These side channels may depend on the secret key, which allows an adversary to gain information of that secret. It is clear, that such an attack is very specific to a concrete implementation of a cryptographic primitive. Famous examples for passive side channels are the power consumption [10], the timing behavior [11], the electromagnetic radiation [12, 13], or even the acoustic emission [14].

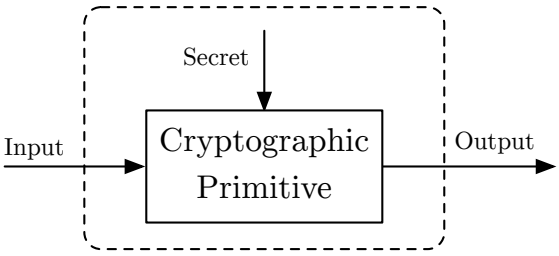



Figure 2.12: Traditional cryptographic system.

Furthermore, an attacker may tamper a cryptographic device to induce faults [15, 16]. The adversary then observes the behavior and the output of the cryptographic system which may allow the attacker to break the system. This kind of attack is called active side-channel attack.

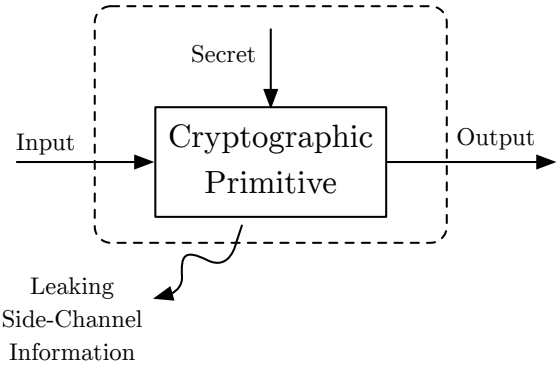



Figure 2.13: Leaking cryptographic system.

### 2.5.1 Simple Power Analysis

With simple power analysis (SPA) attacks, we define the class of attack in which an adversary has a single power trace of a cryptographic operation. Since the power consumption correlates with the internal computation, the cryptographic operation is also detectable in the power trace. Figure 2.14 shows an SPA trace of an AES-128 encryption. One can clearly see the different rounds of the AES operation. In [17] Mangard describes an SPA attack on the key expansion algorithm of AES. If the measurements are too noisy, averaging is exploited to improve the attack. For highly parallel architectures this SPA attack is less likely due to the lower signal-to-noise ratio (SNR).

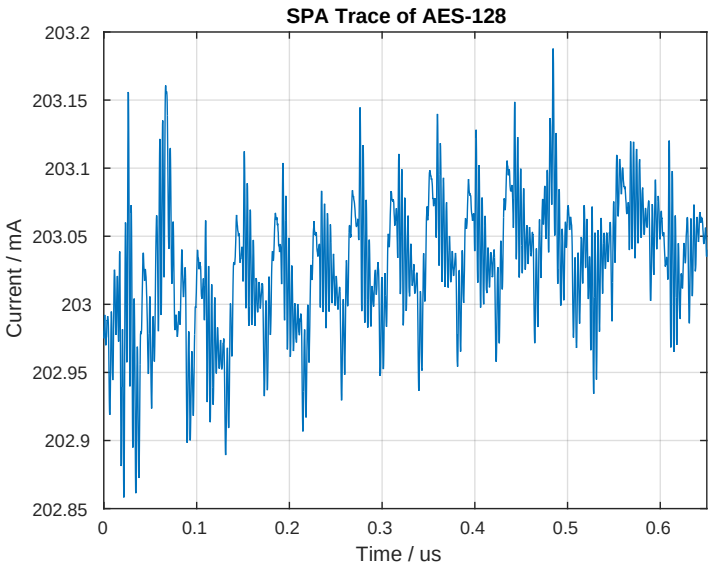



Figure 2.14: AES-128 SPA trace.

### 2.5.2 Differential Power Analysis

The concept of differential power analysis (DPA) was introduced by Paul Kocher [10] in 1999. Compared to SPA attacks, this kind of attack is much more powerful and also much harder to protect. This class of attack combines multiple power traces of encryptions, which all use the same key to recover the used secret. Furthermore, the plaintexts or the ciphertexts used in these traces need to be known by the attacker.

Then the attack works as follows: First, the attacker determines an intermediate value, which depends on the secret and the known values (either the plaintext or the ciphertext). Next, the adversary performs a key guess and computes the intermediate value for this key guess for all power traces. In the next step, the intermediate values are mapped to a hypothetical power consumption depending on a chosen power model. Simple power models such as a simple bit-, Hamming-Weight, or a Hamming-Distance model are often sufficient to model the power consumption accurately. Then a statistical test is performed to compute the relation between the measured power consumption and the hypothetical one. If the key guess was right, a strong correlation between these two power consumptions is visible. However, if the key assumption was wrong, no strong relation between the two power consumptions is shown. This attack is repeated for all possible keys in the key-space to find the one with the strongest correlation. The more accurate the power model is, the fewer power traces are needed to perform the attack.

This principle is used to attack AES-128 successfully. The smallest key for AES is 128 bit wide. Performing a DPA attack on a key-space with the size of $2^{128}$ is not feasible using today's computing power. If the key-space could be reduced, AES can be attacked

more easily. With DPA, the attack is reduced to attack a single key byte, which reduces the key-space to 256 different values. Therefore, a realistic attack is possible. The attack is then repeated for all remaining key bytes but reuses the same power traces.

This algorithm attacks either the first or the last round of AES. In this description we use the first round. The initial step of AES XORs the key bytes of the first round-key (for encryption this equals the cipher key $K$) with the plaintext. Then, the *SubBytes* operation is applied to all results of the XOR operation. This operation sequence is shown in Figure 2.15. The output of the *SubBytes*, the intermediate value $I_i$, is attacked by the DPA attack.

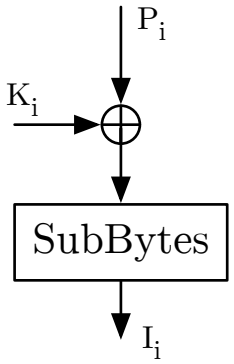



Figure 2.15: Attacked AES-128 value.

For a key guess, the output of the *SubBytes* operation is calculated for all given plaintexts. The result of this operation is mapped to a power consumption depending on a power model. A very simple model is often sufficient for a successful attack. In fact, bit zero of the *SubBytes* output is used as a binary distinguisher between two classes of power consumptions as described in [10]. We now perform the statistical analyze step. Therefore, we compute the mean of both power consumption classes and next the difference of it. If we perform the right key guess, we also perform the right distinguishing step. Therefore, the difference of means shows a correlation. If the key assumption was wrong, the distinguishing step was wrong, and the difference of means does not show a relation between the key guess and the power traces. These steps are repeated for all possible keys for this key byte. Furthermore, this procedure is repeated for all key bytes of the 128-bit encryption key. Using a better power model can enhance the attack to require fewer traces.

Figure 2.16 shows the correlation of a DPA attack on AES-128. The trace shows the correlation result for all key guesses when attacking one single key-byte. The correlation result indicates the right key guess is `166`.

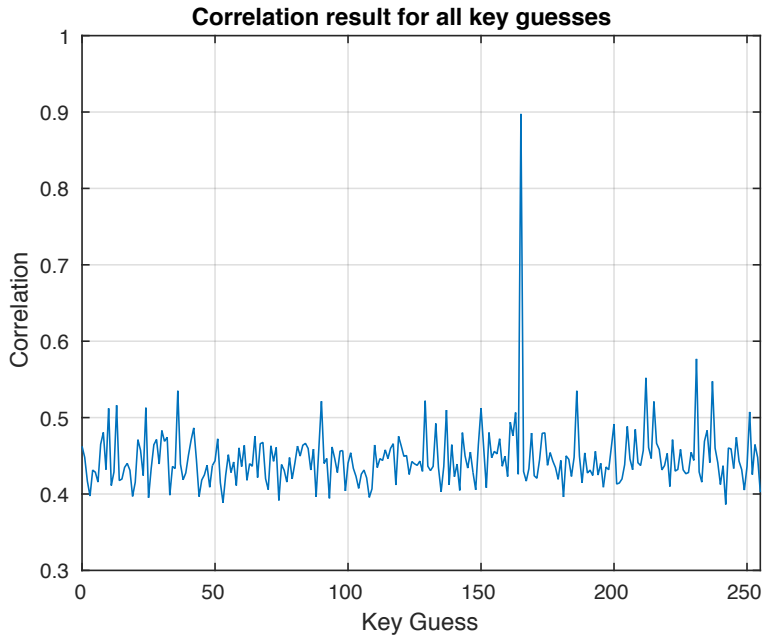



Figure 2.16: Correlation result of AES-128 DPA attack.

#### 2.5.2.1 Higher-Order Differential Power Analysis

In higher-order DPA [18,19] attacks, multiple samples from one trace are combined. The degree of order is defined by how many samples are used from one trace. This extended version of a DPA attack allows an adversary to attack even a masked implementation of a cryptographic algorithm. In a masked implementation, the secret key is masked using a random value. The cryptographic algorithm processes both the mask and the masked value. However, combining the sample points of the mask and the masked value, this again correlates with the secret key. This combining-step, which combines multiple parts of a trace, is a higher-order function.

#### 2.5.2.2 Correlation Power Analysis

Correlation power analysis (CPA) [20] evaluates the correlation between small changes in the traces and a leakage model, which depends on intermediate values of the cryptographic primitive. The Pearson correlation coefficient is used to determine the correlation between a hypothetical power consumption of the intermediate values of the key guess and the actual power traces. Especially when having a lower amount of traces, this variation of a DPA attack might give better results.

### 2.5.3 Fault Attacks

Fault attacks are different to an SPA or DPA attack. An adversary induces faults into the cryptographic algorithm and then observes the behavior of it. Since the attacker is tampering the device, this kind of attack is an active side-channel attack. Boneh et al. showed in [15, 16] how the RSA signature algorithm can be broken by inducing random faults to the RSA algorithm based on the Chinese remainder theorem (CRT). This class of attack is very powerful because breaking the RSA scheme only requires one faulty and one correct computation. Moreover, this kind of attack is also applicable to symmetric ciphers such as AES with the same complexity.

## 2.6 Countermeasures against Side-Channel Attacks

Countermeasures against DPA attacks exist at different levels of abstraction. However, the basic principle is always the same. The countermeasures aim to reduce the SNR to avoid the leakage of secret information. In general, such cryptographic systems are called leakage-resilient. In the backend design of an integrated circuit shieldings [21] are used to avoid probing attacks on the chip. Furthermore, special logic styles [22, 23] exist, which aim to design DPA-resistant standard building blocks. If these building blocks are DPA-resistant, one can build any arbitrary digital circuit, which is DPA-resistant as well. At the algorithmic level, masking is used to break the correlation between the power consumption and the secret. Hiding [24] tries to minimize the SNR to make a DPA attack harder. At the protocol level, fresh re-keying [25] can be used to minimize the number of measurements an attacker can obtain for a single encryption key. All these countermeasures can be used to implement a leakage-resilient cryptographic system.

### 2.6.1 Masking

With masking [26–28] the power consumption becomes independent of the intermediate values to prevent a DPA attack. Therefore, intermediate values are modified using a random mask, which changes for each computation and is unknown to the attacker. This countermeasure operates on the algorithmic level, rather than changing the power consumption of the cryptographic device. It is shown, that using multiple masks (or shares) prevents higher-order DPA attacks. In general, $m$ shares protect against an $m$-th order DPA attack. This countermeasure requires the system to have a random number generator for generating the masks.

### 2.6.2 Hiding

The concept of hiding [24] tries to reduce the dependency of the power consumption and the intermediate values. To achieve this, two approaches can be taken. The first method

attempts to hold the power consumption constant, independently of the processed data. The second idea tries to randomize the power consumption, to avoid a data dependency on it. A popular solution for this countermeasure is shuffling, which randomizes the order of execution of an algorithm. Apparently, this does not solve the data dependency. However, due to a different execution order of the algorithm, the data dependency is shifted. This makes a DPA attack more difficult and requires more traces to successfully mount such an attack.

Chapter 3

# Related Work

Adding countermeasures to an ordinary block cipher can eventually lead to a side-channel resistant implementation. Pramstaller et al. discussed an ASIC implementation of AES-128 [29], which contains masked S-box designs to withstand a DPA attack. The performance loss of such a protected implementation is said to be 40-50 %. However, Mangard et al. [30] published an attack on this ASIC implementation. He demonstrates a DPA attack on the masked implementation succeeded due to glitches in the logic. Although using random masks, the output of the S-box still depends on the processed plaintext. After determining a reasonable power model, a DPA attack was performed with 30,000 traces for a particular S-box used in this design. A comparable performance loss of a masked AES implementation is shown in [31]. The security concerning a DPA attack of 32-bit and a 128-bit masked implementation of AES-128 are compared with their unprotected reference implementation. In both architectures, the throughput is halved compared to the unprotected implementation. Furthermore, the required lookup tables (LUT) for the field programmable gate array (FPGA) are increased by a factor of three.

Instead of implementing an algorithmic countermeasure, Hwang et al. [32] presented countermeasures on the gate level. They published their research on a cryptographic co-processor based on two AES implementations. Both AES designs are functionally equivalent. However, the first architecture contains countermeasures against power analysis attacks. It uses a special logic style called wave dynamic differential logic (WDDL) and a special layout technique called differential routing to counteract DPA attacks. The second AES architecture does not contain any side-channel analysis countermeasures. It was shown that the encryption key was fully recovered after 8000 measurements on the insecure AES implementation. The secure implementation of the co-processor can withstand a full DPA attack even after 1,500,000 measurements. The comparison between protected and unprotected implementation shows the required area increased by a factor

of 2.5. Moreover, the throughput of the protected implementation of AES is reduced by a factor of four.

To avoid adding countermeasures directly to the block cipher, Medwed et al. [25] proposed the concept of fresh re-keying, which is described in more detail in Chapter 4. Their target application is the field of area constraint radio-frequency identification (RFID). They show a low-area implementation of a fresh re-keying system between 8 kGE and 21 kGE. However, there are different constructions of re-keying functions possible. In 2013, Belaïd et al. [33] investigated the construction of a block-cipher-based pseudo-random function (PRF) as a re-keying function. Their focus is to develop a hardware-friendly re-keying function. The paper gives insights on the choice of the components of the block cipher for usage as a re-keying function. However, the construction still needs to be further investigated to provide a good understanding of the security as it is already available for masking and hiding. Further research may lead this construction directly to a leakage-resilient primitive rather than using it for re-keying. Instead of using a re-keying function, which is based on a random nonce, Paul Kocher holds a patent [34] for a direct key update function. Based on a master key $K$ and special tree-functions, a tree of encryption keys is constructed without the need of a nonce. The number of required keys need to be known in advance. However, this allows the architecture to pre-compute the keys to increasing the performance.

Cryptographic implementations are also included in commercial products. Atmel developed the ARM Cortex-A5 processor family SAMA5D4 [35]. This application processor series contains an internal cryptographic unit based on the AES block cipher. The interface of this cryptographic unit supports double-buffering, which means a new configuration can be written while the unit is still busy with its current operation. This AES unit can be combined with a DMA transfer to transparently encrypt or decrypt the memory. Furthermore, the AES unit supports different operation modes including the Galois Counter Mode (GCM). Moreover, the unit can be used to transparently encrypt all memory transfers to the external DDR memory. A second security feature of this processor series is the integrity check monitor (ICM). The ICM is a special DMA controller, which performs an integrity check over multiple memory regions transparently. It computes the hash digest over these regions, which is stored in a dedicated memory range. When reading from a secured memory range, the hash digests are re-computed and compared with the already stored hash values to detect any tampering of the memory.

To further improve the performance of cryptographic implementations, Intel introduced the *AES-NI* instruction set [36] in 2010. This instruction set extension is used to accelerate any AES round-based algorithm. The extension supports executing one round of the AES algorithm for encryption and decryption including a dedicated operation for the last round of AES. Furthermore, a new instruction to accelerate the key expansion is added. When using the newest Intel i7 processor generation named *Skylake*, the performance of the ECB mode is increased to 0.63 cycles per byte (cpb) [37].

However, most of the available commercial products with cryptographic features do not

have any protection against side-channel attacks. There are special fields of applications which offer side-channel protection. Rambus Cryptography Research [38] licences DPA countermeasures to different vendors in the semiconductor industry. Especially smart-cards often contain DPA countermeasures such as shortly described in [39]. However, no concrete countermeasures are presented. To the best of our knowledge, there is no published work on high-performance microprocessor architectures containing countermeasures against side-channel attacks.

Chapter 4

# Leakage-Resilient Cryptography for the Internet of Things

This chapter describes the theoretical background of the cryptographic algorithms used in this thesis. The algorithms are implemented as part of a cryptographic accelerator on a PULP chip. We first discuss the attack model for this thesis. Second we develop the leakage-resilient encryption mode based on AES. Eventually, we describe the permutation-based authenticated encryption framework PASEC.

## 4.1 Attacker Model

The cluster of a PULP chip is considered to be trusted. Everything beyond one cluster is considered to be insecure. This includes the L2-memory, external memory, as well as other communication interfaces. The device contains a secret key $K$, which is unknown to the adversary and is securely stored inside the cluster. It is assumed this key is already in the cluster. This thesis does not consider the secure transfer of the key $K$ into the cluster. The device is deployed in the context of IoT, which means an adversary has physical access to it. It is assumed an attacker can apply non-invasive and semi-invasive passive attacks such as measuring the power consumption, the electromagnetic emission, or the timing behavior. An attacker cannot perform invasive attacks such as probing. The cryptographic device does not have any debug- or test interfaces on which attacks can be performed. However, the attacker can gain information from the information leakage on side channels, which can be obtained in a non-invasive or semi-invasive way. Furthermore, an adversary may induce faults in the cluster. This may be done via glitches on the clock signal.

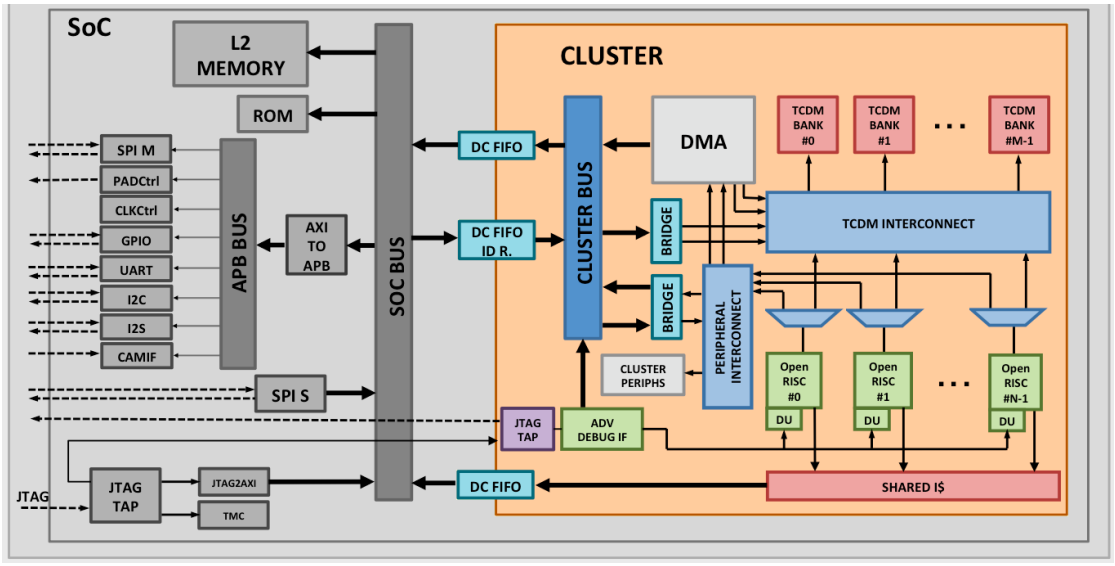



Figure 4.1: Security boundary of the PULP system.

Figure 4.1 shows the PULP system with the cluster inside. The cluster boundary is considered to be the security boundary. The cluster contains three interfaces to other parts of the chip, namely one Advanced eXtensible Interface (AXI) instruction interface, which can only read data, and two AXI interfaces to read and write data, or to control the peripherals of the chip. In this thesis, we focus on the data interfaces. The instruction interface is not protected in this work. The main reason for this is to be as flexible as possible for the evaluation of the actual cryptographic algorithms. It is clear that the encryption unit can be extended to the instruction cache to transparently decrypt fetched instructions.

## 4.2 Fresh Re-keying

A naive way to protect a block cipher against side-channel attacks is to add proper countermeasures to the cipher itself. However, this is difficult to implement due to complex non-linear designs of block ciphers. Implementing side-channel analysis (SCA) countermeasures directly in the block cipher massively increases the area and decreases the performance as reported in [29, 31, 32]. To simplify the protection, Medwed et al. proposed the concept of fresh re-keying. A re-keying function $g$ computes a new, different session key $k^*$ based on one secret master key $K$ and a publicly known random nonce $n_i$, which is shown in Figure 4.2.

Having a different key for every encryption prevents a DPA attack on the cipher by design because a DPA attack requires the encryption key to be constant to be able to observe multiple power traces for one key. However, this scheme shifts the duties of protection

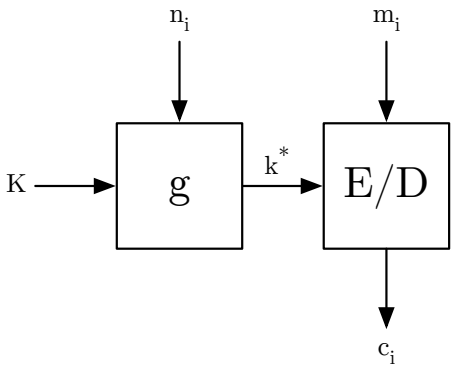



Figure 4.2: Fresh re-keying as proposed by Medwed et al.

against DPA attacks from the cipher to the re-keying function $g$. Due to the simplicity of that function (compared to the block cipher), this function is easier to protect than the block cipher. In a re-keying scheme, the computed session key $k^*$ is only for exactly one encryption. This limits an attacker to gain information on the secret master key under only a single query, which means the attack vector on the block cipher is reduced to an SPA attack. Furthermore, this concept avoids differential fault attacks by design. A differential fault attack requires at least one correct and one faulty encryption or decryption under the same key and same data. Since in the fresh re-keying scheme every encryption happens under a different session key, such an attack is mitigated by design as well. However, the re-keying function $g$ requires special protection against DPA attacks if it is not DPA-safe by design.

At Africacrypt 2010 Medwed et al. proposed this architecture with a new re-keying scheme [25] based on a polynomial multiplication in the finite field $\mathrm{GF}(2^8)[y](y^{16}+1)$. To yield DPA security, the implementation of the polynomial multiplication includes two countermeasures, namely masking and shuffling of the partial multiplications. This function was used to fulfill the requirements for a general re-keying function as defined by the following rules: The re-keying function should have a good diffusion on the master key, in which the bits of the session key $k^*$ depend on many bits of the master key. Furthermore, the re-keying function should be easy to protect against SCA attacks. There should be no synchronization overhead, and the original key should stay the same. These properties need to be taken into account when developing an alternate re-keying function.

## 4.3 Re-keying Function based on a Polynomial Multiplication

In this section, we describe the re-keying function $g$, which is based on a polynomial multiplication in $\mathrm{GF}(2^8)[y]/(y^{16}+1)$. We first start with the plain multiplication and

then describe the added SCA countermeasures. Eventually, this section describes an improvement to avoid the session key to be invertible.

### 4.3.1 Polynomial Multiplication

As described in [25], the polynomial multiplication in $\mathrm{GF}(2^8)[y]/(y^{16}+1)$ fulfills the required properties for a re-keying function. In this multiplication, the 128-bit operands are split into 16 coefficients, each in the finite field $\mathrm{GF}(2^8)$. Because the polynomial multiplication is a linear function, passive side-channel countermeasures such as masking are easily implemented.

In the fresh re-keying scheme described in [25], the polynomial multiplication is performed using the product scan algorithm. However, when considering a parallel implementation of all coefficients in hardware, this increases the hardware overhead due to additional multiplexing. For this reason, the proposed algorithm is transformed into the operand-scan algorithm as described in Algorithm 5. Because all coefficients $a_i$ and $b_j$ are elements in the finite field $\mathrm{GF}(2^8)$, the multiplication and addition in Line 6 of Algorithm 5 operate in this field as well. The finite field is defined by the irreducible Rijndael polynomial [7] $x^8 + x^4 + x^3 + x + 1$.

**Algorithm 5:** Operand scan algorithm for the polynomial multiplication.

**Input**: $a, b \in GF(2^8)[y]/(y^{16}+1)$
**Output**: $c = a \cdot b \in GF(2^8)[y]/(y^{16}+1)$

```
1  j ← rand() (mod 16);
2  i ← 16 − j (mod 16);
3  c ← 0;
4  for k ← 0 to 15 do
5  |   for l ← 0 to 15 do
6  |   |   c_l ← c_l + a_i · b_j;
7  |   |   if l < 15 then
8  |   |   |   i ← i + 1 (mod 16);
9  |   |   end
10 |   |   j ← j + 1 (mod 16);
11 |   end
12 end
13 return c
```

This implementation of the polynomial multiplication already incorporates one countermeasure against side-channel attacks, namely shuffling of the partial multiplications. In Line 1 in Algorithm 5 the start index $j$ is shuffled.

Shuffling the start index results in 16 different sequences of the partial multiplications due to the 16 coefficients. We improve the shuffling strategy to mix all coefficients rather than only the start index. This increases the number of possible sequences of the partial

multiplications from 16 to 16! which equals about $2.09 \cdot 10^{13}$ different sequences. To achieve this, we use the Fisher-Yates Algorithm [40,41], which is described in Algorithm 7. The necessary changes to the re-keying function are shown in Algorithm 6.

**Algorithm 6:** Polynomial multiplication with Fisher-Yates shuffling.

**Input**: $a, b \in GF(2^8)[y]/(y^{16}+1)$
**Output**: $c = a \cdot b \in GF(2^8)[y]/(y^{16}+1)$

```
1 c ← 0;
2 for k ← 0 to 15 do
3 |   j ← fisher_yates_shuffle();
4 |   for l ← 0 to 15 do
5 |   |   i ← l − j (mod 16);
6 |   |   c_l ← c_l + a_i · b_j;
7 |   end
8 end
9 return c
```

The Fisher-Yates algorithm used in Algorithm 6 shuffles an array of $N$ elements, in this case 16. For the polynomial re-keying, we shuffle the current index $j$. For this reason, the array to be mixed is initialized with the values 0 to 15. The Fisher-Yates algorithm is used in an iterative way, which means rather than shuffling the array at once, we use it to iteratively shuffle the array.

**Algorithm 7:** Fisher-Yates shuffling algorithm.

**Input**: Initialized array $a$ with value 0 to 15
**Output**: Shuffled array $a$

```
1 c ← 0;
2 for i ← 15 to 1 do
3 |   j ← Random integer such that 0 ≤ j ≤ i;
4 |   Swap(a[i], a[j]);
5 end
6 return a
```

In hardware, getting a random number in the range $0 \leq j \leq i$ is not possible because we can only retrieve a fixed-size random values between a certain range. Taking the modulus with respect to $i$ is also not possible in most of the cases since this would bias the probability distribution of the shuffled data. However, there are certain cases in which taking the modulus is possible without shifting the distribution. This happens if the current index $i$ evenly divides the number of elements (N=16). If this is not the case, the next random number has to be evaluated until we get one in the desired range. To speed-up the choosing process for the next index, multiple random numbers are evaluated in parallel.

### 4.3.2 Masked Multiplication

The second countermeasure against DPA attacks is masking. With masking, we add a random mask to the key to making the power consumption of intermediate values of the multiplication independent of the secret key. Masking can be applied multiple times to counteract higher-order DPA attacks. Algorithm 8 shows a masked version of the polynomial multiplication described in Section 4.3.1. It uses additive masking to protect the secret value denoted by $a$. All multiplications in this algorithm use the polynomial multiplication from Algorithm 5. Due to the increased number of multiplications ($m+1$ multiplications vs. one multiplication in the unmasked case) this countermeasure decreases the throughput of the polynomial multiplication.

**Algorithm 8:** Masked polynomial multiplication

**Input**: $K, n_i, m_i \in GF(2^8)[y]/(y^{16}+1)$ with $i = 1$ to the masking order $m$
**Output**: $k^* = K \cdot n_i \in GF(2^8)[y]/(y^{16}+1)$

```
1 mk ← K;
2 for i ← 1 to m do
3 |   mk ← mk + m_i;
4 end
5 k* ← mk · n_i;
6 for i ← 1 to m do
7 |   k* ← k* + m_i · n_i;
8 end
9 return k*
```

This masked multiplication algorithm supports an arbitrary masking order without having any memory overhead. Higher-order masking only increases the runtime due to more multiplications.

### 4.3.3 A Non-Invertible Re-keying Function

Dobraunig et al. presented a generic chosen-plaintext key-recovery attack in [42], which is applicable on the re-keying function $g$ as described in Section 4.3. The major problem of this function is that it is invertible. If an adversary can recover a session key $k^*$, he is also able to recover the master key $K$ since the nonce $n_i$ is publicly known. Therefore, Dobraunig et al. refine the requirements for the re-keying function, and add one additional requirement: The re-keying function must be hard to invert. To overcome this issue they describe an improved version of the re-keying function $g$ to make it not invertible. Figure 4.3 shows a feed-forward construction using a block cipher to make the re-keying function non-invertible. Since a block cipher is already used in the bulk encryption scheme, there is not much additional overhead.

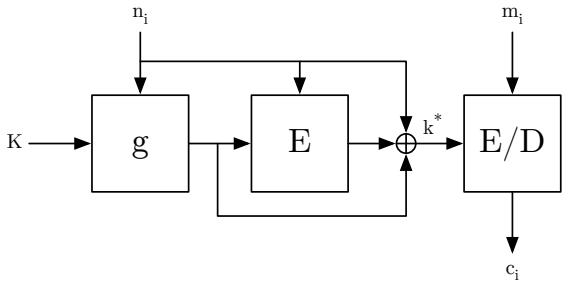



Figure 4.3: A non-invertible re-keying function.

## 4.4 An Efficient Leakage-Resilient Construction

The re-keying scheme from Section 4.3 requires one nonce to encrypt one block of data, which means there is a 100 % memory and communication overhead. Such an encryption scheme is not feasible in practice. Considered the memory encryption scenario, this halves the available memory space because nonces need to be stored as well. In a communication scenario, this halves the throughput, since the nonces also need to be transmitted. Therefore, we aim for a more efficient solution with respect to memory consumption while also keeping the security at the same level.

Figure 4.4 shows a generic encryption pipeline with a re-keying function. The first encryption can already use the fresh session key. To avoid multiple calls of the re-keying function, we aim for an iterative key update function to compute the next session key based on the previous one. The next session key (which is of course only used once) can be used to encrypt or decrypt the next block of data.

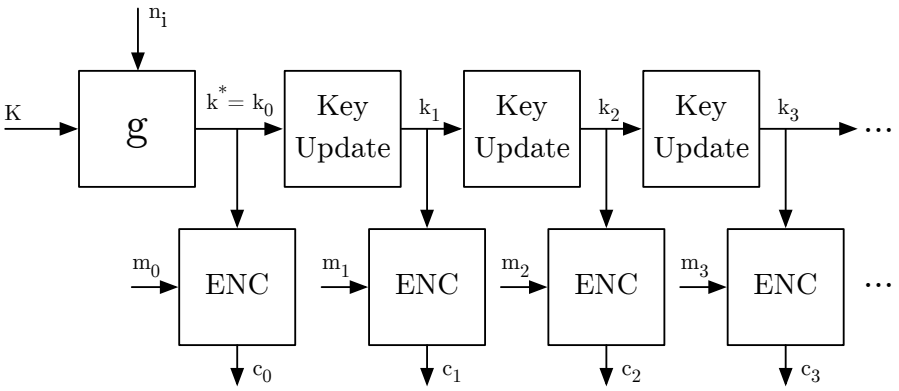



Figure 4.4: Generic encryption pipeline.

The re-keying function is already defined by the masked polynomial multiplication from Section 4.3.2 with the feed-forward construction from Section 4.3.3.

Next, we analyze the requirements for the sequential key update function to find a suitable one. This function ensures that each memory location in one chunk of data is encrypted differently. Given a fresh input key $k_0$, this function derives a different key $k_i$ for each block in one chunk. Since this function is invoked for every memory address in one chunk, it has high throughput requirements and should only have little latency. In general, a lightweight function is preferable. Furthermore, this function should not have a high memory overhead, since the number of executions of this function scales linearly with the amount of data to be encrypted.

### 4.4.1 2PRG Construction

Standaert et al. evaluate in [43] the information leakage of a 2PRG construction , which is illustrated in Figure 4.5. Furthermore, they argue that the previously used „bounded leakage“ model is hard to fulfill in practice, and introduce a more realistic leakage model the so-called „simulatable leakage“ . In this model, the system remains secure if an adversary is unable to distinguish an actual power trace based on the secret key $K$ from a simulated power trace using a random key $K^r$.

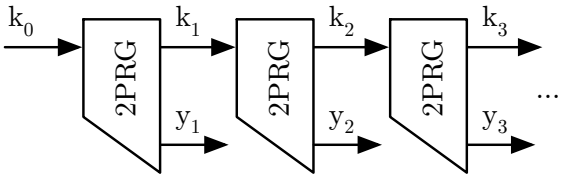



Figure 4.5: 2PRG construction.

Such a 2PRG construction can be implemented using two block cipher instances as depicted in Figure 4.6. In this scheme the values $p_0$ and $p_1$ are public constants. In fact, they show that such a construction is secure when using a block cipher instance which has a 2-simulatable leakage. Simulatable leakage is claimed to be empirically verifiable. However, this requires further investigations as there are still open challenges.

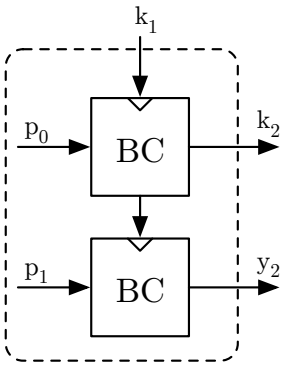



Figure 4.6: 2PRG instance using a block cipher.

The 2PRG construction from Figure 4.6 is turned into a stream cipher. The pad-stream $y_i$, computed by the 2PRG construction, is used to encrypt the message blocks via an XOR operation.

Combining the polynomial multiplication from Section 4.3.3 and the 2PRG construction described above, results in the cipher architecture shown in Figure 4.7. In fact, this architecture is a synchronous stream cipher using the 2PRG construction as a pad generator. Furthermore, the block ciphers in this architecture only need to support the encryption mode rather than both encryption and decryption. For hardware and software implementations, this reduces the required overhead. This allows hardware implementations to have a smaller area footprint and software implementations to have a smaller executable. In this cipher architecture, both ciphers can work in parallel since there are no data dependencies between them.

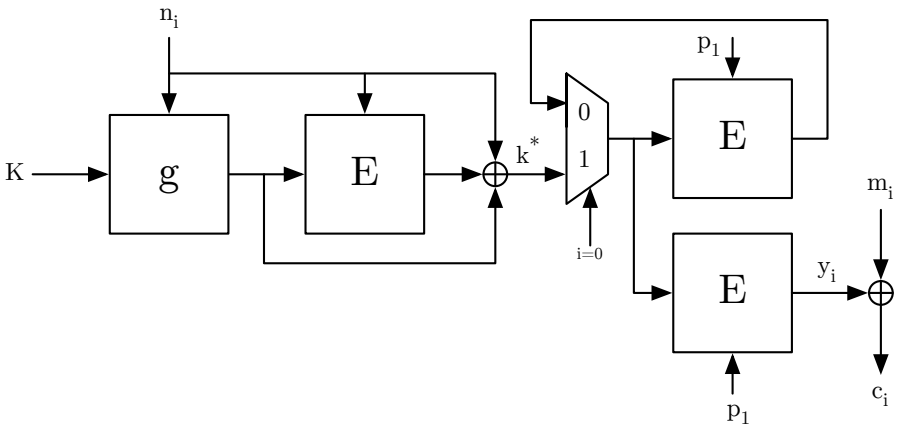



Figure 4.7: Leakage-resilient stream cipher.

For this leakage-resilient 2PRG construction, we use the AES-128 block cipher as it is also used in [43] for their proof. However, the security proof holds for any block cipher which has a 2-simulatable leakage.

## 4.5 Permutation-based Leakage-Resilient Encryption

Another solution to implement a leakage-resilient encryption mode is the authenticated encryption framework PASEC [44], which is based on a permutation function. In general, it follows the same principle as before. It consists of a leakage-resilient re-keying function combined with a bulk encryption mode, which ensures that every message block is encrypted differently. Additionally, this framework supports authentication of the ciphertext to implement an authenticated encryption mode to also withstand DPA attacks.

### 4.5.1 Permutation Leakage

As described in [44], permutation functions support a very flexible modeling of the SPA-leakage in absent of traditional countermeasures such as masking or hiding. Given a bounded leakage of $\lambda$ bits on the internal state, this can be seen as a reduced capacity $c' = c - \lambda$. To keep the same security level, which is determined by the capacity, two approaches can be taken. First, the size of the state of the permutation $p$ can be increased. Second, the rate can be reduced to $r' = r - \lambda$ to retrieve the desired capacity $c$. The latter approach can be used to implement a flexible architecture, in which the rate can be changed dynamically during runtime. If the leakage is too high, the rate is reduced. However, the system needs to be able to deal with different rates.

### 4.5.2 Re-keying Function

PASEC defines two re-keying functions named *RK1* and *RK2*. Both re-keying functions are DPA-secure by design and do not contain additional countermeasures such as masking or hiding.

#### 4.5.2.1 Re-keying Function RK1

This re-keying function is based on the Goldreich-Goldwasser-Micali (GGM) [45] construction. First, the state is initialized using the zero-padded master key $K$ followed by an application of the permutation function. Next, the GGM tree is constructed, by either taking the left half or the right half of the current state, depending on the current bit $n_{i,l}$ of the public nonce $n_i$. The remaining half of the state is padded using the current index $i + i$. After constructing the next state, the permutation function is applied. This is repeated for all bits of the nonce to finally compute the session key $k^*$. Figure 4.8 visualizes this computation.

#### 4.5.2.2 Re-keying Function RK2

A second approach to construct a permutation-based re-keying function is to use directly the sponge construction, which is depicted in Figure 4.9. Here the state is initialized using the master key $K$ padded with a dedicated initial vector $iv$, followed by an application of the permutation function. Then all $l$ bits of the nonce $n_i$ are iteratively absorbed into the state. This is followed by an application of the permutation function to finally compute the session key $k^*$.

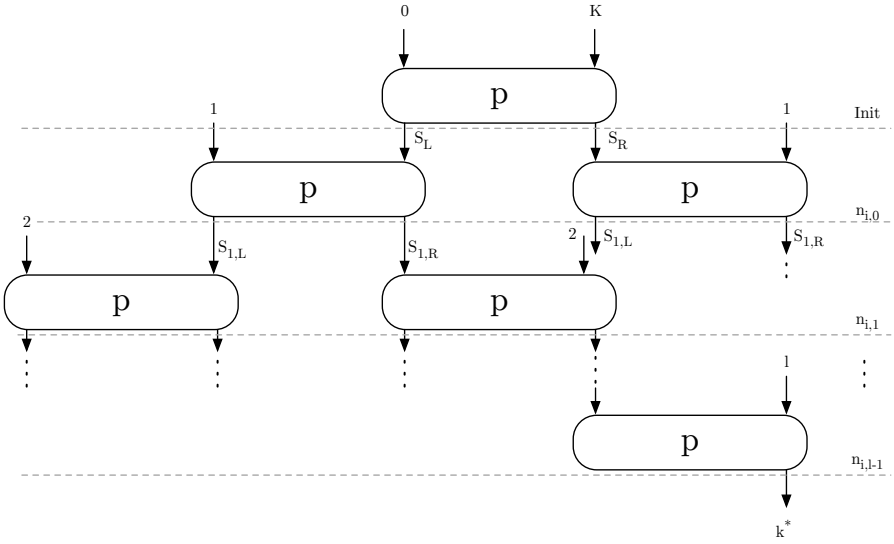



Figure 4.8: Re-keying function *RK1*.

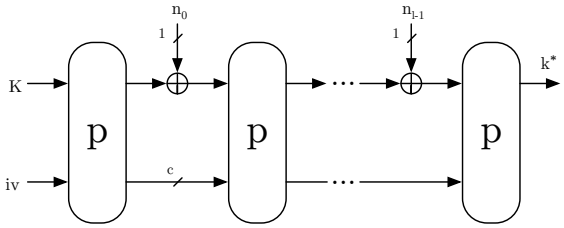



Figure 4.9: Re-keying function *RK2*.

### 4.5.3 Encryption Function

The encryption mode in PASEC is constructed using a classic sponge construction. The state of the permutation function is initialized using the obtained session key $k^*$ in combination with the nonce $n_i$, and a dedicated initial vector for encryption $iv$. The session key $k^*$ is computed either via the re-keying function *RK1* or *RK2*. After performing an initial application of the permutation function, the rate-part $r$ of the state is used to generate a pad-stream for encryption, which is depicted in Figure 4.10. The generic re-keying function $g_1$ can either be the re-keying function *RK1* or *RK2*.

### 4.5.4 Authentication Function

PASEC also defines an authentication mode in an encrypt-then-MAC construction. As illustrated in Figure 4.11, first, the ciphertext data is hashed to get the value $y$. The value $y$ is then used to compute a MAC-key $k_2$, which depends on the ciphertext. For this purpose the generic re-keying function $g_2$ is either the re-keying function *RK1* or

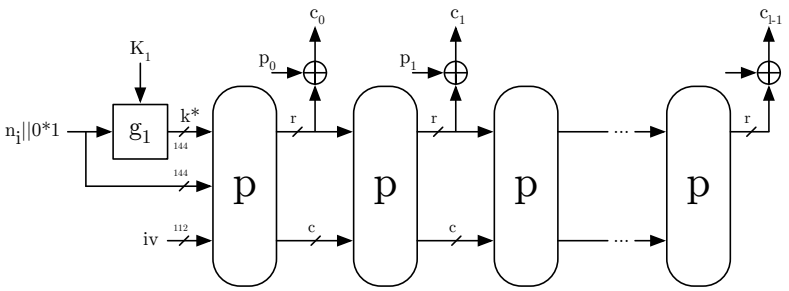



Figure 4.10: Sponge construction for encryption.

*RK2*. This key $k_2$ is finally used to compute the authentication tag $T$. Although this authentication algorithm supports associated data, which would be hashed before the ciphertext, this is not used in the application of the PULP architecture.

The encrypt-then-MAC construction is required because otherwise, it would open a door to a DPA attack. In a MAC-then-encrypt construction the authentication tag is computed on the plaintext and is then encrypted with it together. In the verification step while decrypting, the ciphertext and the tag are first decrypted before the integrity is checked. In this case, an attacker may tamper the memory to keep the value of the nonce constant. This would allow the adversary to capture multiple power traces using the same master key $K$ and the same nonce $n_i$ but with different data, which eventually leads to a DPA attack.

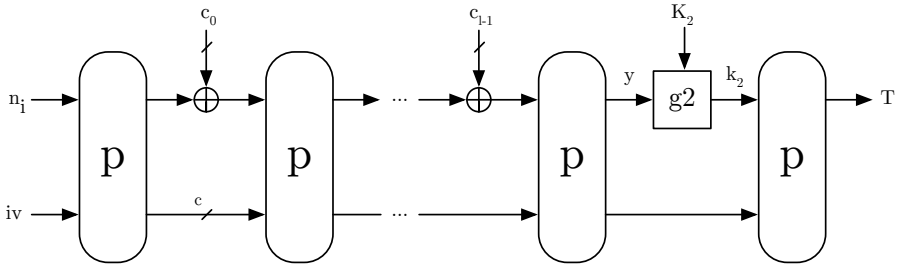



Figure 4.11: Sponge construction for authentication.

### 4.5.5 Parameter Selection

Permutation-based cryptographic modes support a flexible trade-off between performance and security. For all operating modes, we require a security of 128-bits. Let us define $b$ with the size of the permutation $p$, $k$ the size of the master key $K$, and $t$ the size of the authentication tag $T$. Furthermore, we denote the size of the capacity of the permutation with $c$ and the size of the rate of the permutation with $r$. Using these definitions, PASEC defines the security bounds for the particular primitive functions *RK1*, *RK2*, ENC, and MAC in Table 4.1. For the targeted security level of 128-bit, we use the KECCAK-$f$[400]

permutation as described in Section 2.4.1. KECCAK-$f$[400] uses a state size of 400-bits, which is sufficient to reach a security level of 128-bit according to Table 4.1.

Furthermore, [1] also defines the required number of rounds of the KECCAK-$f$[400] permutation to achieve 128-bit security given a 128-bit master key $K$. Contrary to the original definition of KECCAK-$f$[400] in [9], which requires 20 rounds for this permutation, PASEC relaxes the required number of rounds for the re-keying function *RK1* and for the MAC algorithm. The required number of rounds for PASEC is summarized in Table 4.2.

## 4.6 XTS Encryption

We want to compare the leakage-resilient modes described before with state-of-the-art cryptographic modes used for disk encryption systems. For this reason, we also implement the XTS [46] mode of operation, which is used in various commercial products [47,48].

XTS stands for XEX-based tweaked-codebook mode with ciphertext stealing, which is a recommended mode of operation by the National Institute of Standards and Technology (NIST). This scheme is based on the XOR-Encrypt-XOR construction [49] and uses the AES [7] as its encryption primitive. Compared to XEX, XTS uses two different keys $K_1$ and $K_2$ for the tweak derivation and the encryption. The construction of XTS is shown in Figure 4.12

Although this mode of operation was developed for disk encryption, it is also suitable for the use case of memory encryption, or any arbitrary communication encryption. Disks are usually organized in larger blocks, the so-called sectors. This concept also applies to the main memory. Instead of directly reading one single value from the main memory, the cache controller fetches one cache line, which can be considered as the sector. For a communication scenario, a message number may be seen as the sector number to derive a different tweak.

As shown in Figure 4.12, the sector number (SN) is used to derive a sector-dependent tweak. Furthermore, a multiplication in the finite field GF($2^{128}$) with $\alpha^{addr}$ computes a different, address-dependent tweak for each block of memory in one sector. The address-dependent tweak is used to whiten the input and output of the block cipher. Due to

Table 4.1: Security bounds for the PASEC framework [1].

| Function | Security (bits) |
|---|---|
| RK1 | $\min(k, b/2, c)$ |
| RK2 | $\min(k, c/2, t)$ |
| ENC | $\min(k, b/2, t)$ |
| MAC | $\min(k, b/2, c, t)$ |

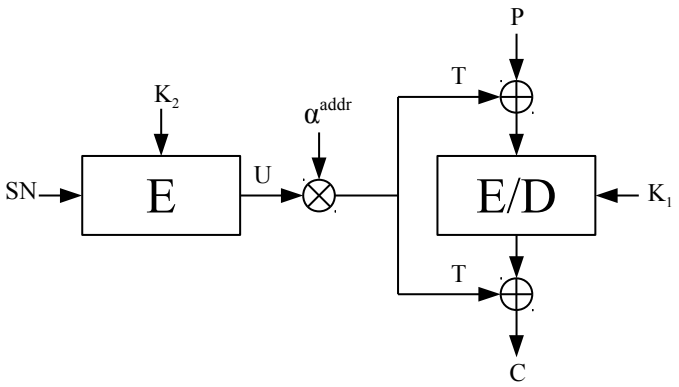



Figure 4.12: XTS encryption mode.

different tweaks for each address in the sector, each value gets encrypted differently. Therefore, this construction mitigates weaknesses from ECB, in which the same plaintext always encrypts to the same ciphertext. However, the XTS construction is not SCA resistant by default. To do so, the block cipher, as well as the tweak derivation function, need to include countermeasures against this kind of attacks.

Table 4.2: Recommended parameters for PASEC [1].

| Function | Capacity $c$ | max. State Leakage | Rounds |
|---|---|---|---|
| RK1 | 256 | 128 | 12 |
| RK2 | 256 | 128 | 20 |
| ENC | - | 72 | 20 |
| MAC | 399 | 271 | 12 |

Chapter 5

# Hardware Architecture

This chapter describes the hardware architecture of *HWCrypt*, a cryptographic accelerator for the PULP system. We describe each relevant component of the design by starting with the top-level architecture. Details of sub-components are then given in a top-down approach. Furthermore, we present the functional verification flow used in this design. This covers the verification using an RTL-flow, as well as a software-defined verification of the accelerator.

## 5.1 Accelerator Architecture

There are different possibilities to achieve the goal of a secure communication off the cluster. An encryption unit could be added directly at the communication bus level, which was done in [50] for the AXI interface. Such a design supports a transparent encryption for all memory transfers on the bus. Adding a confidential communication over an input/output (I/O) interface is hard to achieve in such a design. This type of communication is fairly different from memory accesses. Furthermore, such an architecture does not support a flexible evaluation of the cryptographic algorithms.

A second possibility is adding an encryption module in the DMA controller of the cluster. DMA is used to transfer larger blocks of data from the L2-memory to the TCDM, and vice versa. Due to bigger block sizes, one can gain better performance and efficiency but is still limited to DMA transfers only. Achieving an I/O encryption in this scenario requires a special DMA controller for I/O within the cluster.

A third approach is implementing a dedicated cryptographic accelerator, which operates as a co-processor from the CPU perspective. The processing cores can program the accelerator to encrypt data from the TCDM and then start an operation. The accelerator operates in parallel to the processing cores and notifies them when it finishes. After

finishing one encryption operation, the CPU core can program the DMA controller to transfer the encrypted data off the cluster, or send it via an I/O interface outside the chip. For this thesis, we choose this approach. The accelerator supports both the encryption of the TCDM inside the cluster and the communication of data sent over an I/O interface off the chip. Furthermore, this approach allows us to flexible evaluate of the cryptographic operations concerning the DPA safety. Additionally, this design gives us the possibility to use the internal primitives in software. This allows us to implement primitive-based algorithms in software, but with hardware acceleration.

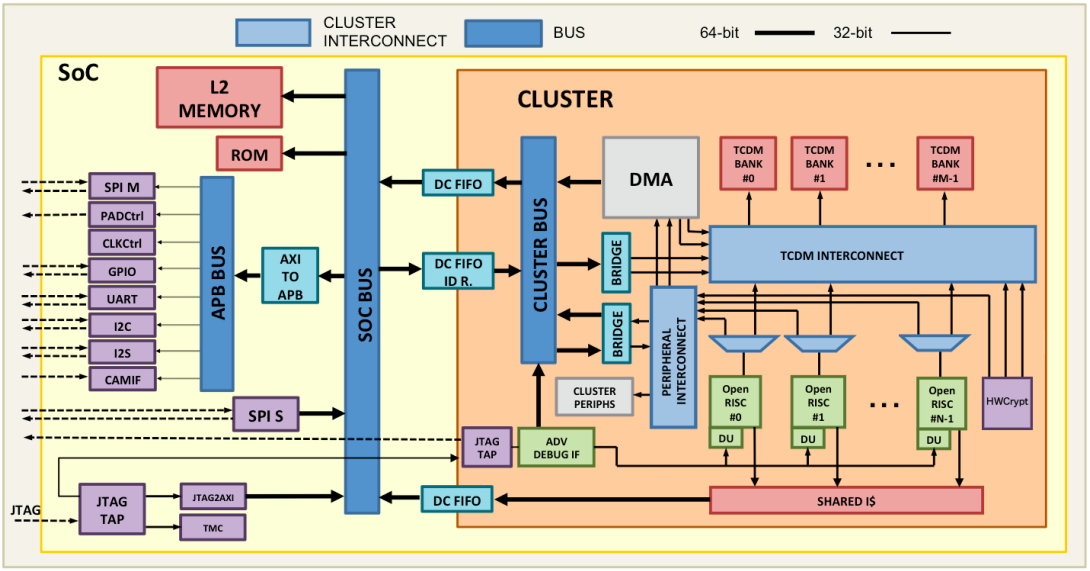



Figure 5.1: PULP architecture with *HWCrypt* accelerator.

Figure 5.1 shows the PULP system with the cryptographic accelerator *HWCrypt* inside the cluster. This is indicated by the purple unit next to the OpenRISC processing cores. The connection shows the TCDM interface of the accelerator, and a peripheral interface, which the processing cores use to program the accelerator.

## 5.2 Operation

The main goal of the accelerator is to encrypt and decrypt data of the TCDM using algorithms, which resists side-channel attacks. Chapter 4 presents the algorithms supported by this accelerator. It describes two leakage-resilient encryption schemes, which both rely on the principle of fresh re-keying. For a better flexibility, we require the accelerator also only to perform the re-keying operation alone. The freshly computed session key can then be used in software for further processing. Third, we require the accelerator to allow a primitive operation. This means the inner primitive operations of the encryption schemes – the AES round function and the permutation function – should be accessible

via the accelerator interface. This can be used to implement any algorithm in software, which is based on these primitives, but with hardware acceleration.

## 5.3 HWCrypt - A Cryptographic Accelerator

Figure 5.2 shows the top-level architecture of the *HWCrypt* accelerator. The accelerator is part of the PULP cluster and, therefore, provides one peripheral interface. This interface allows the processing cores to configure, start, and monitor the accelerator. In addition to the peripheral interface, *HWCrypt* facilitates two 32-bit TCDM interfaces, which are used by the accelerator to load and store the ciphertext and plaintext from the TCDM. Both TCDM interfaces are used only in one direction, which means that one interface only reads data, whereas the other one only writes data to the TCDM. Furthermore, *HWCrypt* contains two interrupt or event lines, which are used to notify the processing cores that the operations of the accelerator have finished.

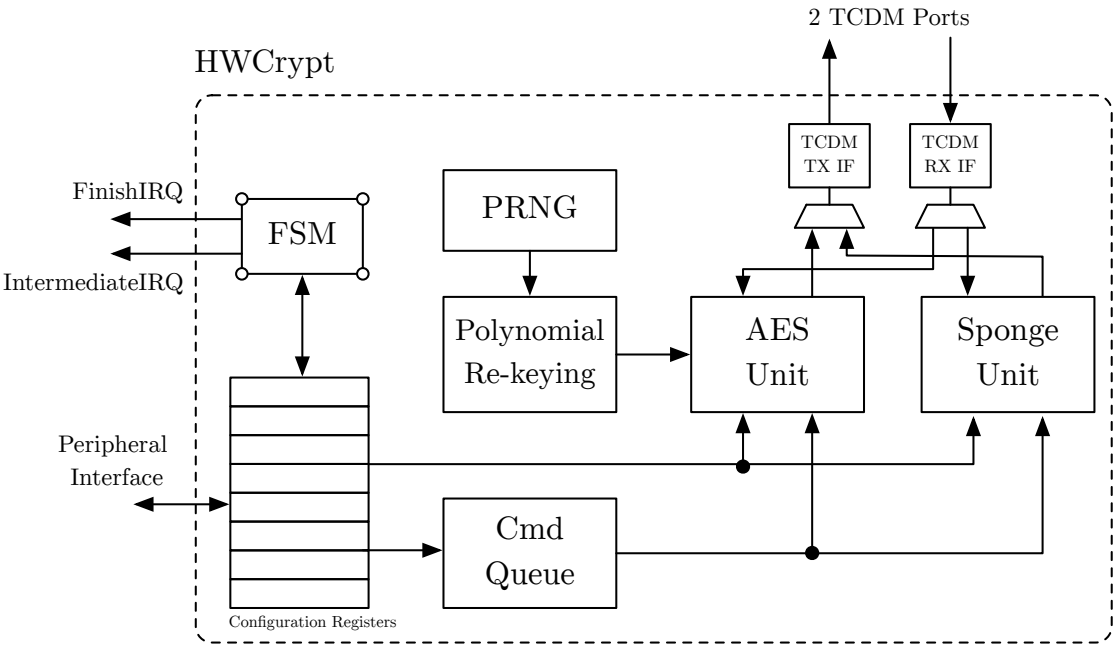



Figure 5.2: Top-level architecture of the *HWCrypt* accelerator.

Internally, the accelerator uses the *AESUnit*, the *Polynomial Re-keying Unit*, and the *SpongeUnit* for computing the cryptographic operations. Moreover, the design uses a finite state machine (FSM), two TCDM interface controllers, a command queue, and a register interface for the data transfer and the control flow. The design rationales of these components are now discussed in detail.

## 5.4 Peripheral Interface

A 32-bit peripheral interface is used for the register configuration. The accelerator registers are visible for the CPU cores in a certain memory range and can be read or written by the processing cores. These memory accesses are transferred into accesses to the peripheral interface. The peripheral interface only supports 32-bit accesses in hardware, which needs to be considered when performing an access via software. A detailed register map is describing all registers of the accelerator, as well as the detailed interface description, are shown in Section A.6 and Section A.5.

## 5.5 Command Queue

The accelerator can only compute one job at a time. However, to improve its efficiency, we implement a command queue. This allows the software to configure multiple pending jobs while the accelerator is still busy with its current operation. Therefore, we define a special set of registers, namely the *queue* registers. These registers are writeable while the accelerator is still busy. Write accesses to all other registers block until the accelerator is in the idle state again. Furthermore, new jobs can be started although *HWCrypt* is still working. When the accelerator finishes one job, it automatically fetches the next pending job from the command queue and starts it. Adding a new job to the queue only works if the queue is not full. However, if the queue is full and the software tries to start a new job, this access is blocked until there is space in the queue again. The main reasons to implement a command queue are the long configuration and interrupt latency of the PULP system. To overcome this drawback, the processing cores can re-program the accelerator while it is still busy. The performance improvements due to adding a command queue are discussed in Section 6.6.

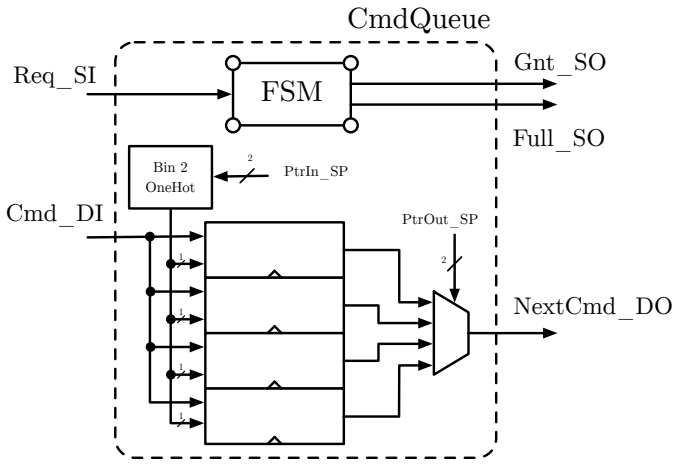



Figure 5.3: Architecture of the command queue.

Figure 5.3 shows the architecture of the command queue, which contains four sets of queue registers. These queue registers are a collection of several different registers, which are required for a new job.

## 5.6 Polynomial Re-keying Unit

In this section, we evaluate different architectures of the masked polynomial multiplier as described in Algorithm 8. This unit multiples two polynomials in the finite field $\mathrm{GF}(2^8)[y]/(y^{16}+1)$. Furthermore, this unit implements additive masking of the secret key as a countermeasure against a DPA attack. We discuss different architectures of the underlying polynomial multiplier, which are based on Algorithm 5.

### 5.6.1 Parallel Masked Polynomial Multiplier

A parallel masked polynomial multiplier consists of multiple unmasked polynomial multipliers to implement a masked version of the algorithm. This does not affect the throughput because all unmasked multiplications are computed in parallel. Such an architecture requires $m+1$ instances of a polynomial multiplier to implement $m$-th order masking, which would increase the area requirements tremendously.

Second, this design is not safe against a DPA attack. Consider first-order masking without shuffling. Such a design requires two polynomial multipliers. The first multiplier performs the computation $m_0 \cdot n_i$. The second multiplier computes the masked multiplication $(K \oplus m_0) \cdot n_i$. In this architecture one register contains the value $m_0$ and another one the register $K \oplus m_0$. The Hamming-distance between those registers equals the key $K = (K \oplus m_0) \oplus m_0$. Assuming a Hamming-distance based power model, this would leak information on the key $K$ via the power consumption. For this reason, this parallel architecture is not suitable for a secure implementation of the masked polynomial multiplication.

### 5.6.2 Iterative Masked Polynomial Multiplier

The iterative masked polynomial multiplier computes all unmasked multiplications iteratively. Since there are no parallel computations of two multiplications, no information leaks via the Hamming-distance are possible by design. Since all polynomial multiplications are computed iteratively, the throughput is decreased linearly with the masking order. One advantage of this architecture is its support of an arbitrary masking order without a major implementation overhead. Figure 5.4 shows the architectural block diagram of the iterative masked polynomial multiplication. This design supports masking up to order 255.

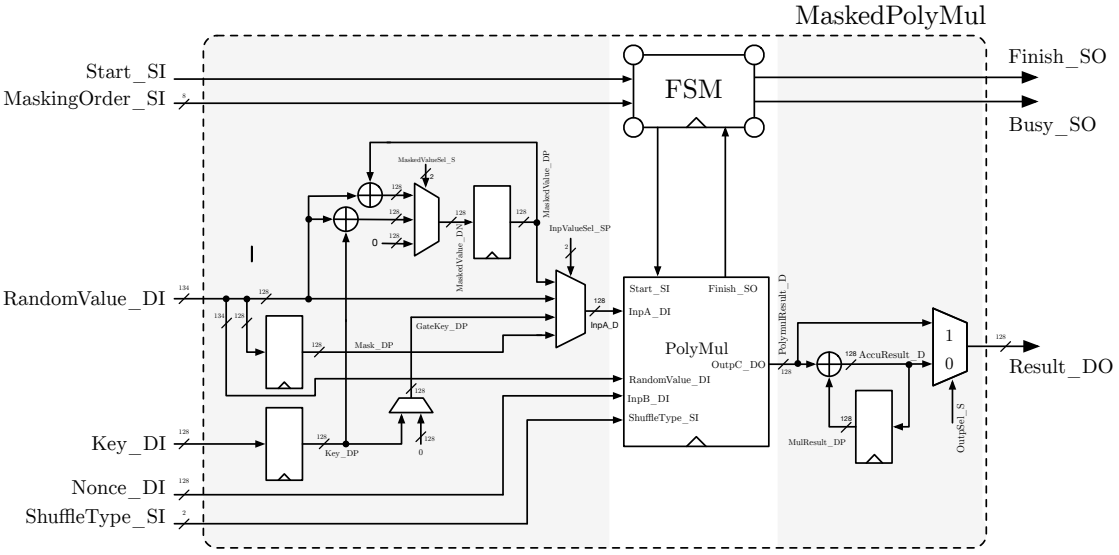



Figure 5.4: Masked polynomial re-keying function.

The architecture can be broken into three parts: on the left, the design contains an accumulator to compute the final masked key. In the middle, the unmasked polynomial multiplier is used to perform any multiplication. On the right part, the output accumulator is shown, which computes the final result of the session key $k^*$. If masking is disabled, the result directly from the polynomial multiplier is used for the output.

**Masked Key Accumulator.** The masked multiplication algorithm, as describes in Algorithm 8, first computes the multiplications between the masks $m_i$ and the nonce $n_i$. Second, the multiplication of the masked key and the nonce is performed. To compute the masked key, the register `MaskedValue_DP` first stores the value $K \oplus m_0$, which implements first-order masking. The polynomial multiplier then computes the multiplication $m_0 \cdot n_i$. In the next step, the next mask $m_1$ is processed. This mask is added to the `MaskedValue_DP` register to get the second-order masked key $K \oplus m_0 \oplus m_1$. Then the mask $m_1$ is processed to compute the value $m_1 \cdot n_i$. These steps are repeated until the desired masking order is reached.

**Output Accumulator.** The output accumulator fetches the results of the unmasked polynomial multiplier and accumulates them in register `MulResult_DP`. For the following description we consider first-order masking. For this operation, first, the result of $m_0 \cdot n$ is stored. Second, the result of $(K \oplus m_o) \cdot n_i$ is added to the output accumulator register to eventually compute the final result of the session key $k^* = K \cdot n_i$.

### 5.6.3 Polynomial Multiplier

The masked polynomial multiplier uses one instance of an unmasked polynomial multiplier, which is described in this section. This unit implements the operand-scan multiplication scheme described Algorithm 6. This polynomial multiplier exploits different shuffling strategies as the second countermeasure against DPA attacks.

#### 5.6.3.1 Fully Parallel Multiplier

A fully parallel multiplier computes all 256 partial multiplications in parallel. Therefore, this architecture requires 256 instances of a multiplier in the finite field GF($2^8$), which increases the area of the multiplier. This design gives the best throughput since the multiplication is computed in one cycle. However, this design does not allow shuffling since all partial multiplications are computed in parallel.

#### 5.6.3.2 Fully Iterative Multiplier

A fully iterative multiplier only consists of one multiplier in GF($2^8$). All 256 partial multiplications are computed iteratively. Due to the iterative computation, shuffling is possible. However, the throughput of this design is worse because it takes 256 clock cycles to compute one multiplication. Since the polynomial multiplier only uses one multiplier in GF($2^8$) the area consumption is low, which makes this design attractive to area-constraint environments such as smart cards.

#### 5.6.3.3 Iterative Parallel Multiplier

The iterative parallel multiplier combines the fully parallel architecture with the iterative design. The goal is to improve the throughput compared to the fully iterative architecture but still support shuffling.

As indicated in Algorithm 6, only the order execution of the outer loop is randomized, which means all operations of the inner loop can be parallelized. This results in the design as proposed in Figure 5.5. This architecture consists of 16 parallel multipliers in GF($2^8$) to implement the inner loop of the operand-scan algorithm. Furthermore, the design consists of 16 multiply-accumulate stages, each containing a multiplier in the finite field GF($2^8$). These stages compute all 16 result coefficients in parallel by accumulating their partial multiplication results. To compute all 256 partial multiplications, this design requires 16 clock cycles since all output coefficients are computed in parallel.

The shuffle unit selects the current coefficient index of the first operand of the multiplication, which is the same for all 16 multiplication stages. The second coefficient index differs for each multiplication stage but depends on the first index. Therefore, this value

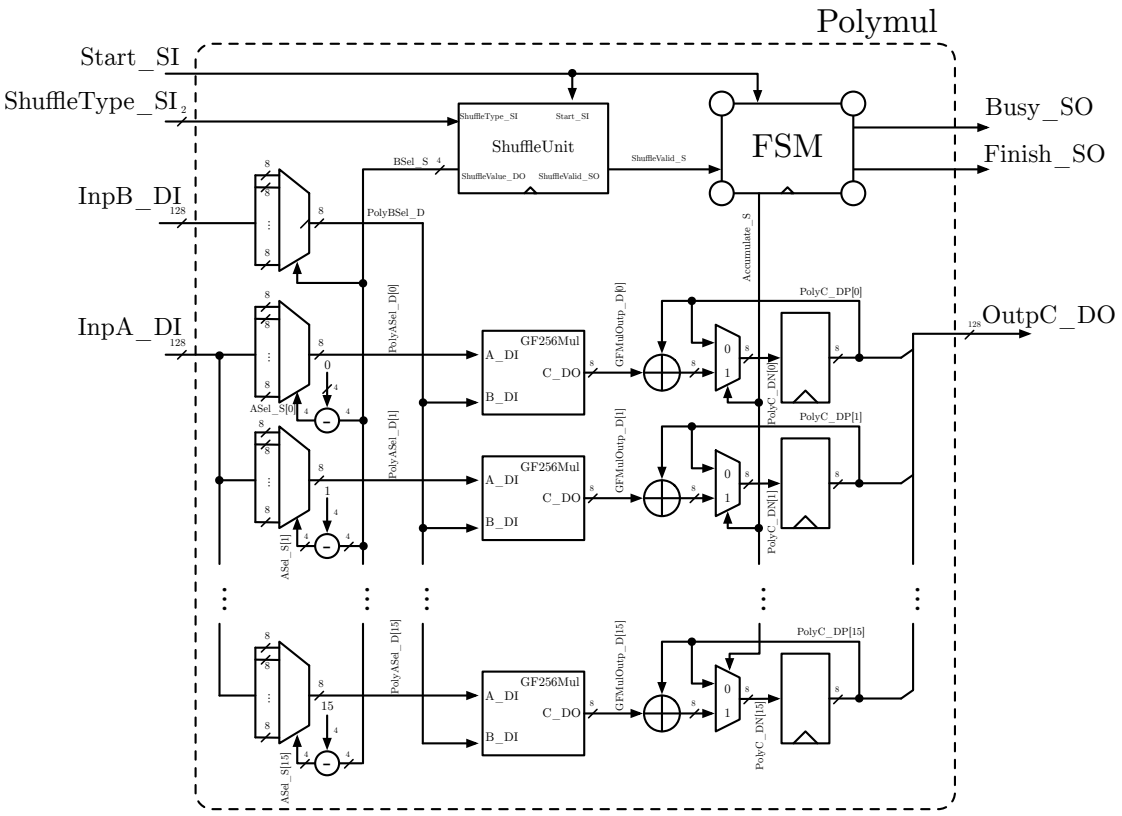



Figure 5.5: Iterative polynomial multiplier.

is computed combinationally using the relation $ASel[i] = i - BSel$. In this relation, $BSel$ indicates the index generated by the shuffling unit. $ASel[i]$ denotes the index for operand $A$ for the $i$-th stage.

In addition to the full Fisher-Yates shuffling, the shuffle unit also supports shuffling only the start index as originally proposed by Medwed et al. Furthermore, shuffling can be completely disabled for evaluation purposes.

## 5.7 Linear Feedback Shift Register

The masked polynomial re-keying unit from Section 5.6 requires random masks for its computation. Furthermore, the polynomial multiplier from Section 5.6.3 requires random data for the internal shuffle unit. We use a linear feedback shift-register (LFSR) to implement a pseudo-random number generator (PRNG) to provide the required data.

The masked multiplication requires 128-bit random masks. The shuffle unit requires, at least, 4-bits of random data per cycle. In the first step of the polynomial multiplication,

both a mask and the random shuffle data are required. Therefore, we require 132-bit of random data in this step. As a result, we implement an XNOR-based 132-bit LFSR based on the polynomial $x^{132} + x^{103}$ [51]. For the sake of simplicity the architecture depicted in Figure 5.6 shows the computation of only one random bit. This schematic is replicated 132 times to compute one random 132-bit vector per clock cycle.

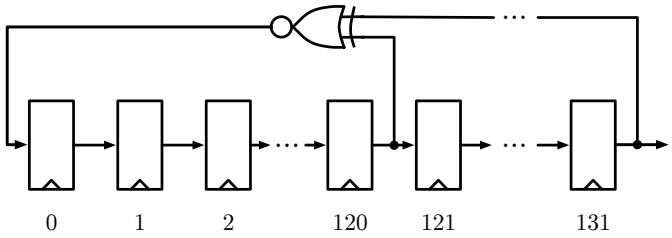



Figure 5.6: 132-bit LFSR architecture.

## 5.8 AES Unit

The AES unit is developed to implement all AES-based encryption modes. This includes the leakage-resilient encryption mode described in Section 4.4.1 and the XTS encryption mode described in Section 4.6. Furthermore, this unit supports the post-processing of the session key as described in Section 4.3.3 and plain ECB encryption using AES-128 for comparison purposes.

Figure 5.7 shows the architecture of the AES unit. The main part of this architecture is one instance of an AES-128 algorithm, which supports computing two words in parallel. This parallel operation of two words is used in XTS- and in ECB-encryption. Moreover, this unit facilitates a 128-bit data input-, and output interface with an AXI-like handshake. The handshake protocol is based on a *ready* and *valid* signal similar to the AXI interface protocol. The design is broken into three parts, namely the input sampler, the AES-128 instance, and the output sampler.

**Input Sampler.** The input sampling unit samples the 128-bit input data from the TCDM interface into two registers `Inp0_DP` and `Inp1_DP`. XTS- and ECB-encryption requires this for a parallel computation of two words. This unit operates independently of the main control FSM of the AES unit as it tries to fetch data whenever it is possible.

**AES-128 Dual Iterative.** The middle part of the AES unit contains one *AES-128 Dual Iterative* instance, which is described in detail in Section 5.8.1. For XTS encryption, the AES unit supports whitening of the input- and output-data by XORing them with an address-dependent tweak. For all other modes, the tweak is constantly kept zero to not modify the input- and output-data. The architecture of the tweak derivation is

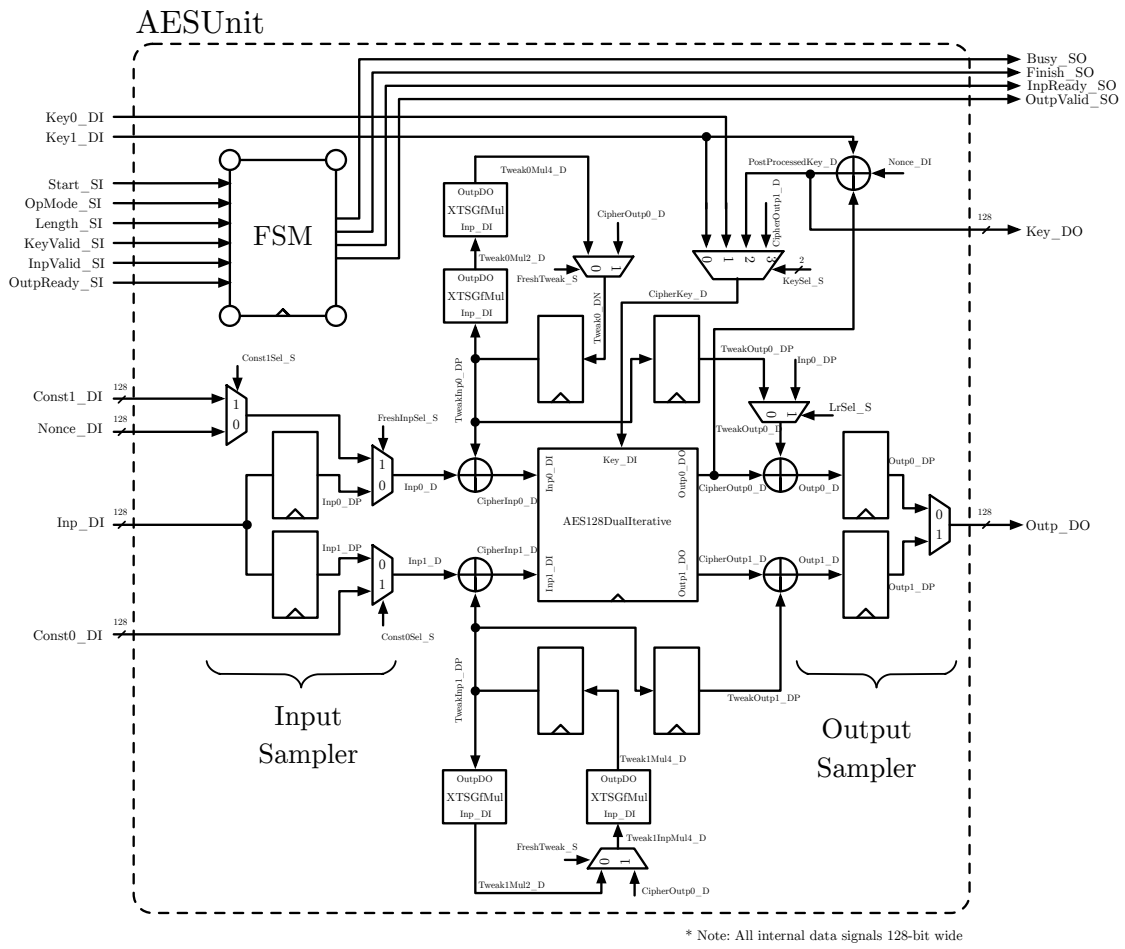



Figure 5.7: Architecture of the AES unit.

described in Section 5.8.2. Since the same tweak is used to whiten both the input and the output of the AES computation, the tweak for the first cipher stage is stored in the registers `TweakInp0_DP` and `TweakOutp0_DP`. This structure is replicated for the second cipher stage.

**Output Sampler.** The AES-128 instance outputs in ECB-, and XTS-mode two result words per encryption since both cipher stages are used in parallel. To output this data over a single 128-bit AXI-like interface, this data is first cached in the output sampling stage defined by the registers `Outp0_DP` and `Outp1_DP`. The output sampler then outputs one by one word on the output interface as this is ready. Meanwhile, the AES-128 instance can already compute the next encryption.

Clearly, if the architecture should only support the leakage-resilient mode of operation, different parts of the design are not necessary. The input- and output-sampler are not

needed since the AES unit operates only on one word rather than on two words in parallel. Furthermore, the GF($2^{128}$) multipliers and the required registers to store the tweaks for XTS encryption can be omitted.

### 5.8.1 AES Dual Iterative

For an efficient implementation of the leakage-resilient AES-based encryption mode as defined in Section 4.4.1, we aim for an architecture which is capable of encrypting two words in parallel. Since both encryptions use the same cipher key for encryption, the key expansion algorithm is shared between both cipher stages. As shown in Figure 4.6, pipelining cannot be exploited since there are data dependencies between each computation. Concretely, the new cipher key for the next encryption depends on the output of the last encryption. Exactly this dependency does not allow us exploiting pipelining in hardware to improve the throughput. These design rationales lead to following design decisions:

1. An iterative architecture of AES
2. Two cipher stages for a parallel computation of two AES words
3. A shared key expansion algorithm between both cipher stages

The design decisions lead to the architecture *AES-128 Dual Iterative*, an iterative design with two cipher stages. Figure 5.8 shows the block diagram of the top-level architecture of *AES-128 Dual Iterative*. The design consists of two cipher stages, each using the round-keys computed by the shared instance of the key expansion algorithm. Each of the cipher stages computes one encryption.

#### 5.8.1.1 Cipher Stage

In Figure 5.9 the architecture of the cipher stage containing two combinational round functions is shown. The number of round functions increases the length of the critical path and is constraint by the target frequency. The second round function is different because this instance supports the last round function as defined in Algorithm 1. This particular function adds a second execution path, which omits the *MixColumn* operation. Having two combinational round functions, the invocation of the ten rounds of AES-128 is achieved in five clock cycles.

For evaluation purposes, Figure 5.10 shows a different architecture of the cipher stage, which consists of three combinational round functions in the critical path. To apply the round function ten times, this architecture requires four clock cycles. In the last clock cycle the round function is applied only once. Therefore, this architecture implements the instance for the last round of AES as the first round function in the combinational path.

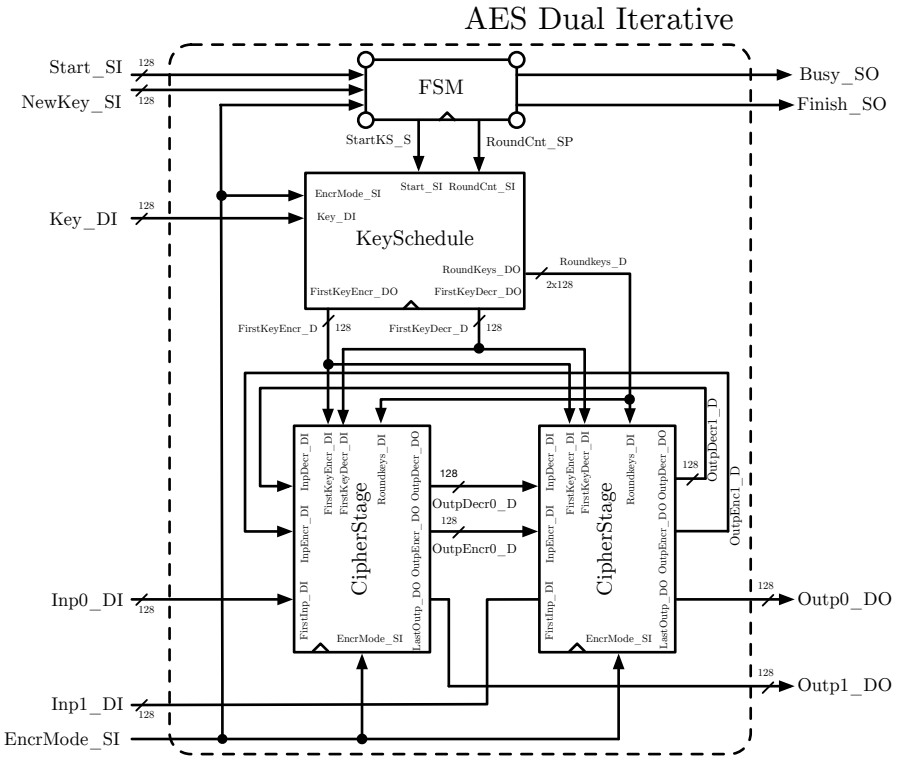



Figure 5.8: *AES-128 Dual Iterative* top-level architecture.

#### 5.8.1.2 Key Expansion Algorithm

Figure 5.11 depicts the key expansion algorithm of AES-128 for a cipher architecture with two cipher stages. The key expansion algorithm supports computing the round-keys for encryption and decryption. The round-keys for encryption and decryption only differ in their order. The computation of them is based on the initial cipher key. A round function for the key expansion algorithm is iteratively applied on the previous round-key as described in Algorithm 3.

The decryption algorithm of AES-128 requires the same round-keys as for encryption, but in the reverse order. Two approaches are considered to support this functionality. First, the architecture contains a memory for all 11 round keys. The key expansion algorithm for encryption is invoked, which stores all round-keys in the memory. For the decryption operation, the pre-computed round-keys from the memory are returned in the reverse order using a multiplexing network.

The second approach only saves the last round-key of encryption. Since the key expansion algorithm of AES is invertible, one can compute previous round-keys given the last one. Therefore, the first step is to perform the encryption variant of the key expansion algorithm to determine the last round-key. This key is then used to compute again

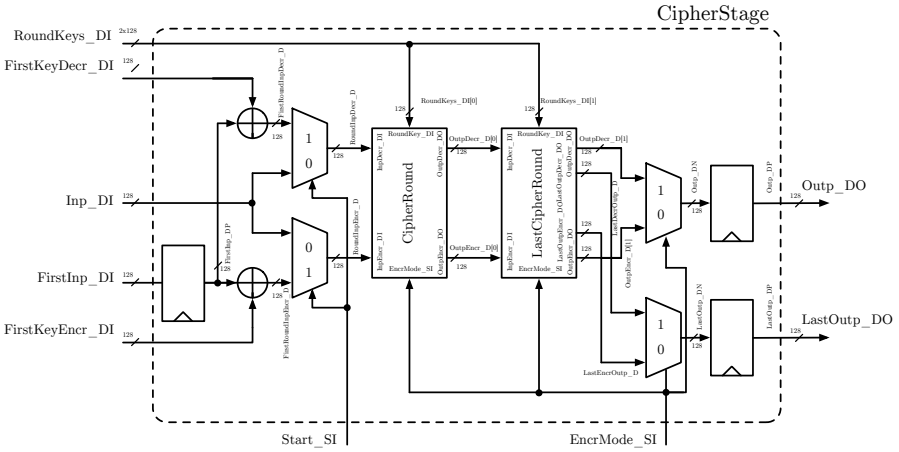



Figure 5.9: Architecture of a cipher stage with two combinational round functions.

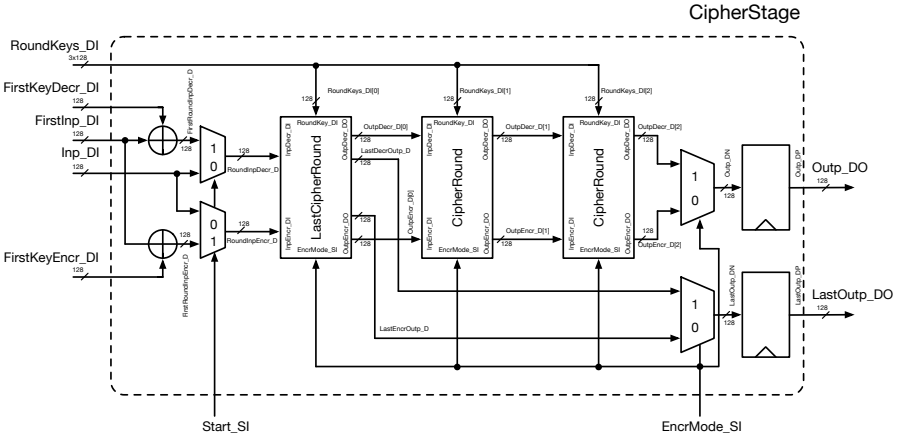



Figure 5.10: Architecture of a cipher stage with three combinational round functions.

the previous round-keys by applying the inverse key update function. This approach is preferable since it requires less memory to store all round-keys. Furthermore, the multiplexing overhead is reduced since the output of the round function is directly used as the round-key for decryption. This architecture is also more flexible because it scales with the number of cipher stages of the cipher.

Figure 5.11 shows the block diagram of the key expansion algorithm for a two-round cipher architecture. This design can easily be extended to a three-round cipher architecture by adding a third key round function in the combinational path. For decryption, the last round-key is stored in the `KeyDecr_DP` register, which is used to compute the previous round-keys using the inverse key update function.

Figure 5.12 shows the parallel design of the round function of the key expansion algorithm of AES-128. This design operates on round-keys organized in four 32-bit words. As

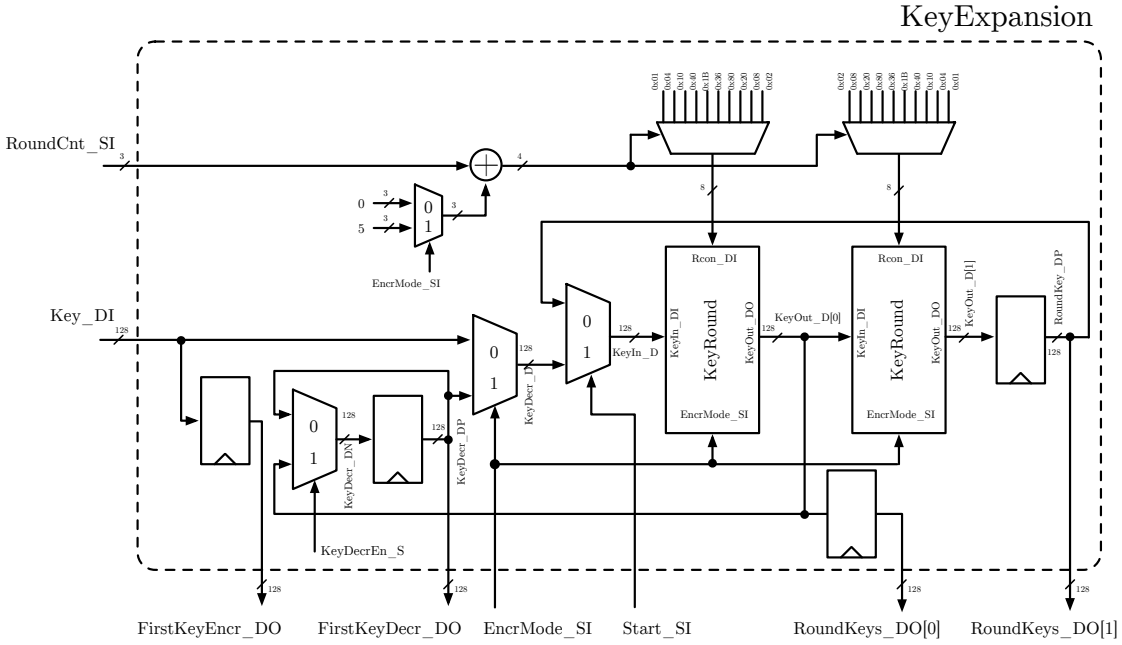



Figure 5.11: Architecture of the key expansion algorithm.

described before, the architecture supports both the forward- and the backward-path of the key expansion algorithm.

#### 5.8.1.3 AES Round Function

The round function for the cipher stage combinationally computes one round of the AES algorithm. Figure 5.13 shows the round function for the last round. This design supports computing the intermediate round as well as the last round. For the sake of simplicity, we only show the design of the last round since the intermediate round function is almost the same, except it does not contain the exit path for the last round. Figure 5.13 shows the architecture of the round function with its two signal paths for encryption and decryption. Furthermore, it shows the exit paths for the last round, which omits the *MixMatrix* operation.

**SubMatrix.** The *SubMatrix* operation performs the S-box lookup for each of the 16 bytes of the state. Depending on the `EncrMode_SI` signal, the lookup is either performed for encryption or decryption. We evaluated two different algorithms for the S-box lookup. As described in Section 2.3, the mathematical description of the S-box lookup is based on a multiplicative inverse in the finite field GF($2^8$) followed by an affine transformation. The inverse S-box lookup performs the opposite operation using an inverse affine transformation followed by the multiplicative inverse. The multiplicative inverse is computed

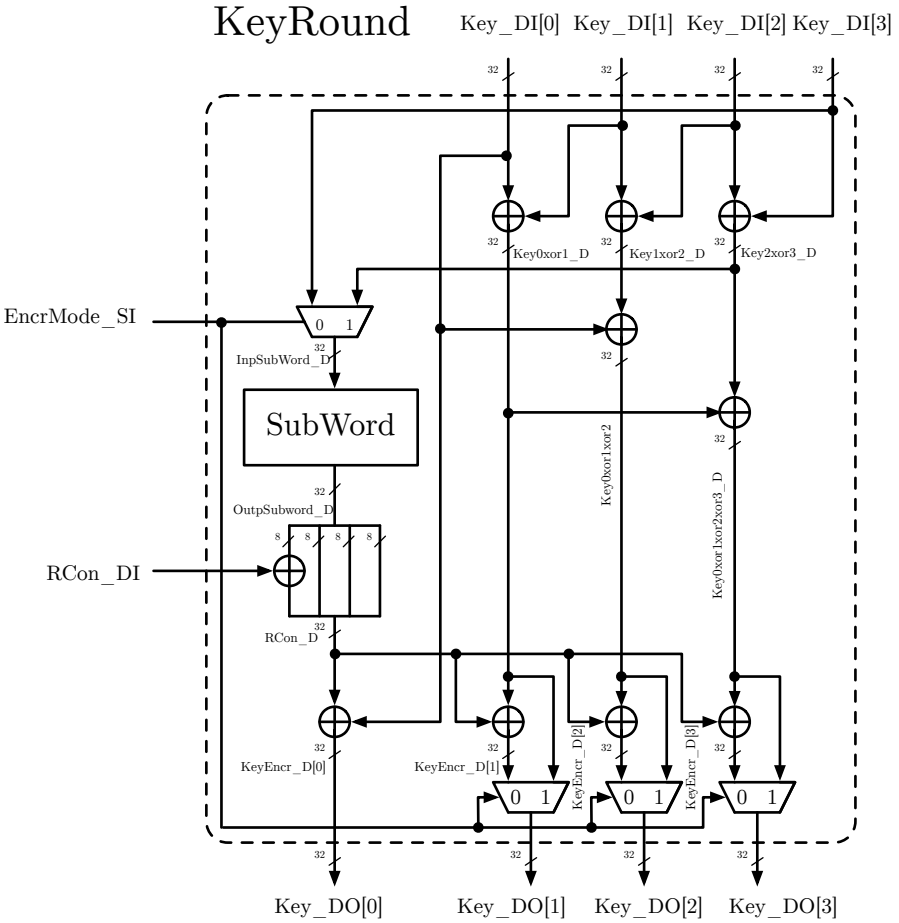



Figure 5.12: Architecture of the round function of the key expansion algorithm.

using a lookup in a table (LUT), which is shared between encryption and decryption. The affine- and inverse-affine transformations are computed combinationally. This design is denoted as the *shared* design. The second approach is using a dedicated LUT for both encryption, and decryption. Both LUTs contain the multiplicative inverse as well as the required pre-, or post-transformation. Compared with the first approach this architecture requires two LUTs. However, this design is faster since no combinational transformation and multiplexing are required. We name this architecture as the *separate* design. The architecture of both approaches is depicted in Figure 5.14.

**ShiftRows / InvShiftRows.** This operation performs a cyclic shift of the state. This only contains a rewiring of the state in hardware.

**MixMatrix / InvMixMatrix.** The *MixMatrix* operation performs the *MixColums* step on all bytes of the state. The design uses an optimized 32-bit implementation as

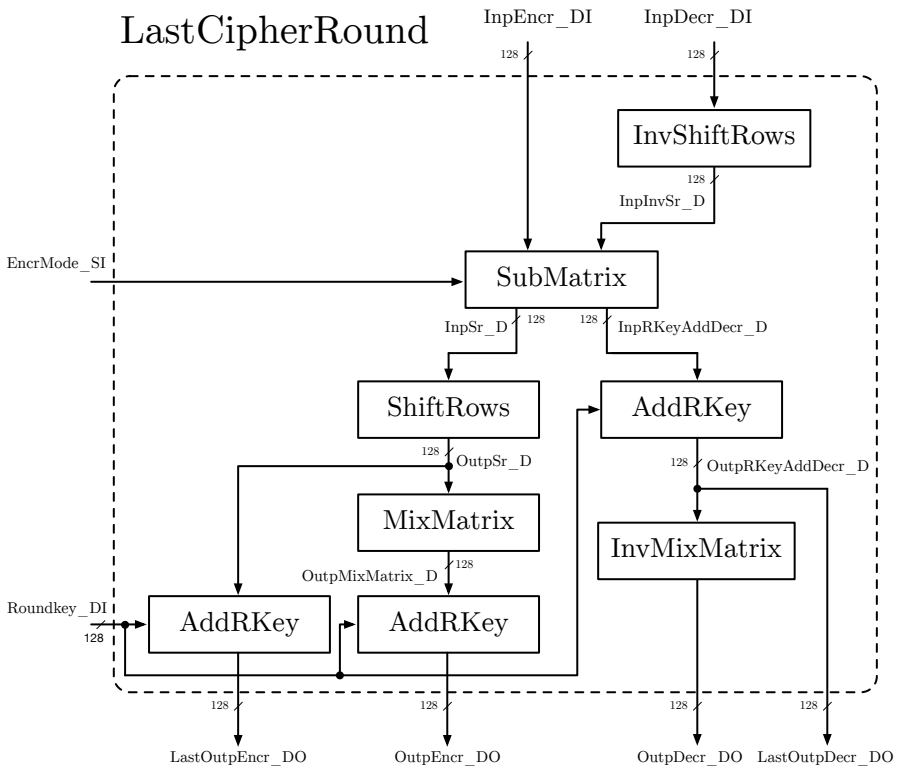



Figure 5.13: AES last round architecture.

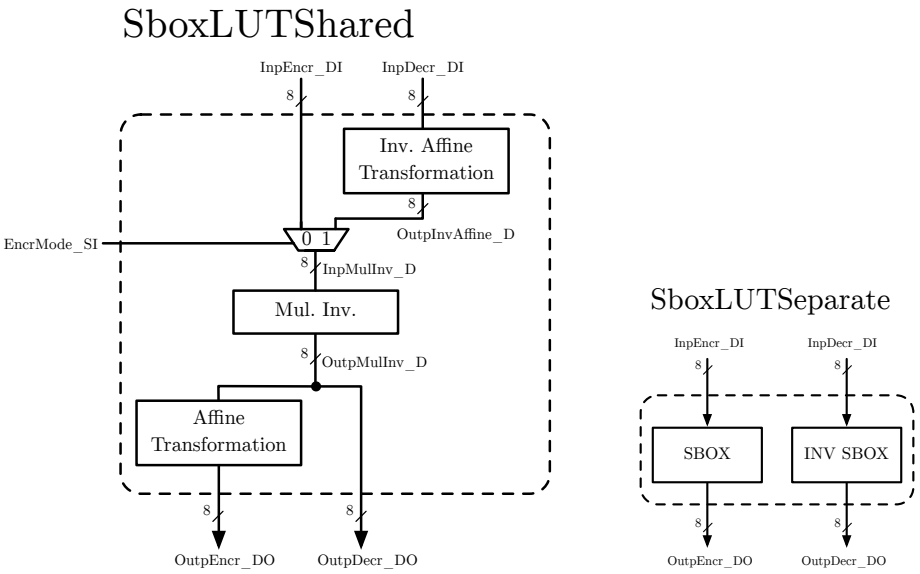
proposed in [52]. Furthermore, this unit also supports the inverse operation for the decryption operation.

**AddRkey.** The *AddRkey* performs an XOR operation between the state and the round-key for the particular round. Adding the round-key works the same for encryption and decryption.

#### 5.8.1.4 Architecture Evaluation

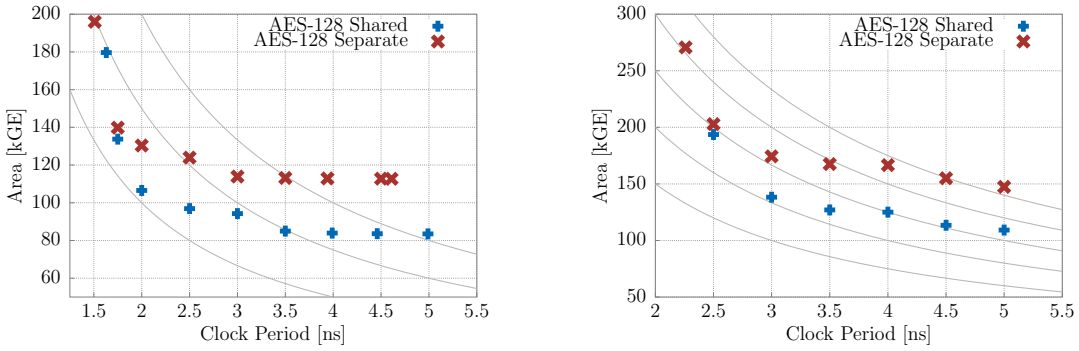
For comparison reasons both S-box, architectures are synthesized for the targeted UMC 65 nm technology. Figure 5.15 shows the Area-Time-plots (AT) of different synthesis runs for both the two- and the three-round architecture. Additionally, the plots also show the synthesis results for the two different S-box architectures.

As assumed, the separate S-box design is faster than the shared one. Due to the longer critical path as shown in Figure 5.14a the shared design is slower than the separate design shown in Figure 5.14b. However, since the latter architecture requires in total two LUTs for encryption and decryption, the area requirements are higher.




(a) Shared. (b) Separate.

Figure 5.14: Two AES S-box designs.




(a) 2-round. (b) 3-round.

Figure 5.15: AT-plots of different synthesis runs for two AES-128 architectures.

The target frequency for synthesis of *HWCrypt* is 500 MHz since previous tape-outs of PULP chips in the same process technology indicate a maximum clock frequency of 500 MHz under the typical-case conditions (VDD = 1.2 V, T = 25 °C). For tape-out, the design is synthesized using the worst-case process corner (VDD = 1.08 V, T = 125 °C) libraries, which are significantly slower than the typical-case libraries. To have a reasonable area of *HWCrypt*, the target frequency for synthesis is reduced to 400 MHz.

Figure 5.15b shows the AT-plot of the three-round architecture of AES-128 using the typical-case parameters. Since the minimal clock period of this design is close to 2.5 ns, the two-round architecture is favorable. In Figure 5.16 we show the AT-plot of the two-round architecture using the worst-case process parameters. Compared to the typical-case synthesis run, the minimum clock period is significantly reduced to 2.49 ns. The plot also indicates the 400 MHz threshold at a clock period of 2.5 ns.

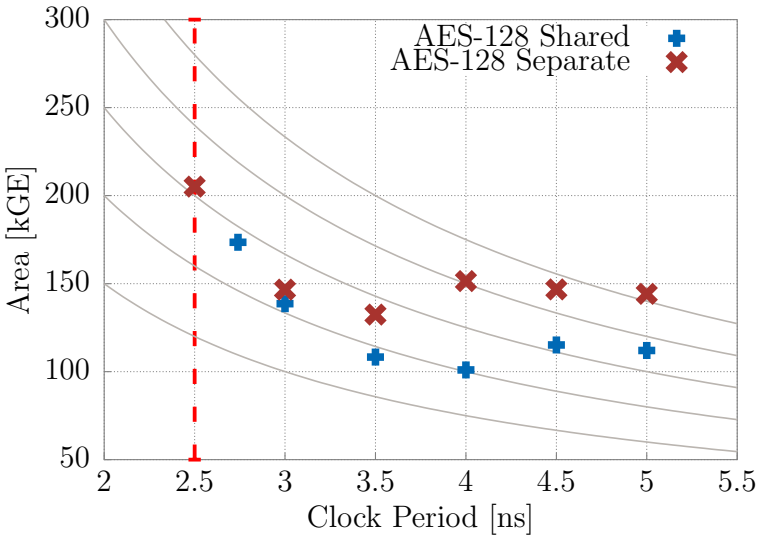



Figure 5.16: AT-plots of two-round AES-128 using worst-case parameters.

The AT-plot shows the results for both the shared, and the separate S-box design. For the *Fulmine* tape-out, we use the separate design to reach the target frequency of 400 MHz when synthesizing with the worst-case parameters.

### 5.8.2 GF($2^{128}$) Multiplier

XTS encryption requires a sequential tweak update based on a multiplication in the finite field GF($2^{128}$) defined by the polynomial $x^{128} + x^7 + x^2 + x + 1$. The initial tweak is derived from the sector address and is denoted by $T_b$. Address-dependent tweaks within one sector are computed according to Equation 5.1.

$$T_{addr} = T_b \cdot \alpha^{addr} \tag{5.1}$$

Intuitively this would require an exponentiation function to compute the address-dependent tweak. The complexity can be reduced. First, $\alpha$ is a constant. The recommended parameters for $\alpha$ are 2 or 3 [46]. In this work we use the value $\alpha = 2$. Second, this tweak derivation function computes the tweak for each memory address within one sector. Equation 5.1 computes this address-dependent tweak always from the initial base tweak. This is turned into a sequential derivation function as shown in Equation 5.2. The next tweak is always computed from the previous one. This turns the exponentiation function into a simple multiplication by 2 in GF($2^{128}$).

$$T_i = T_{i-1} \cdot 2 \tag{5.2}$$

Multiplication by two, although in the finite field GF($2^{128}$), is fairly simple since it only consists of a shift by one to the left with a conditional XOR with the irreducible polynomial. The hardware architecture of this multiplier is shown in Figure 5.17.

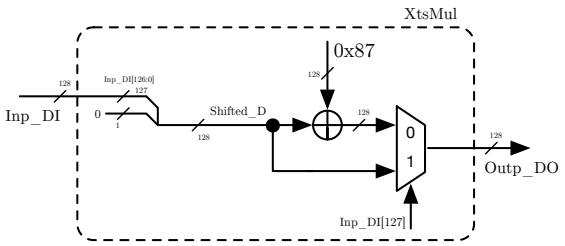



Figure 5.17: GF($2^{128}$) multiplication by 2.

## 5.9 Sponge Unit

The sponge unit implements a versatile architecture to support the permutation-based re-keying functions *RK1* and *RK2*. Furthermore, this unit supports a flexible encryption mode as well as authentication of the ciphertext. The sponge unit, as depicted in Figure 5.18, is based on two instances of the permutation function Keccak-$f$[400] as described in Section 5.9.2. The first permutation function is used for re-keying, encryption, and parts of the authentication algorithm. The second permutation is used for the

authentication part during encryption, in which authentication and encryption can work in parallel. Due to the parallel computation, the throughput is increased. Similar to the AES unit, this design facilitates a 128-bit data input- and output interface with an AXI-like handshake protocol.

The architecture of the sponge unit shows an iterative construction using a permutation function on which all algorithms are based on. Both permutation functions use an input multiplexer controlled by an FSM to select the right state input. The output of the permutation function is fed into the *Variable Rate Engine* to absorb or squeeze data with the correct rate. An iterative design is used because we cannot exploit pipelining since there are data dependencies between each computation of the permutation function.

### 5.9.1 Variable Rate Engine

To support a flexible trade-off between performance and security, the sponge construction supports a variable data rate. The permutation function uses the variable rate engine only to operate on smaller parts of the data. This unit supports processing the following rates: 1-bit, 2-bit, 4-bit, 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit.

Figure 5.19 shows the hardware architecture of the *Variable Rate engine*. This design is based on two major parts. First, the unit is consists of a shift-register with a variable shift-width for all supported rates. Second, there is an XOR unit which XORs the lower $r$-bits of the internal data `D` with the state `State_DI`. The parameter $r$ denotes the supported rates as stated before.

### 5.9.2 Keccak-$f$[400] Permutation

The permutation-based re-keying and encryption modes use the Keccak-$f$[400] permutation function. This permutation function consists of 20 iterative round functions, as described in Algorithm 4. Figure 5.20 shows the architecture of a three-round version of Keccak-$f$[400]. For 20 invocations, during the last clock cycle, only two round invocations need to be invoked rather than three. Therefore, the state is exited after two rounds and multiplexed at the output register. Moreover, to support a more flexible evaluation of the security of the permutation-based algorithms, the implementation of Keccak-$f$[400] supports less than 20 computations of the round function. In fact, the architecture supports any round number, which is a multiple of three, up to a maximum number of 20 rounds.

An iterative approach is chosen since pipelining is not exploitable. Due to data dependencies from the output to the input, pipelining the architecture would not gain any throughput improvement. The number of combinational rounds is defined by the desired maximum frequency of 400 MHz using the worst-case process parameters. In Figure 5.21 we compare two the synthesis results of the three-round architecture using

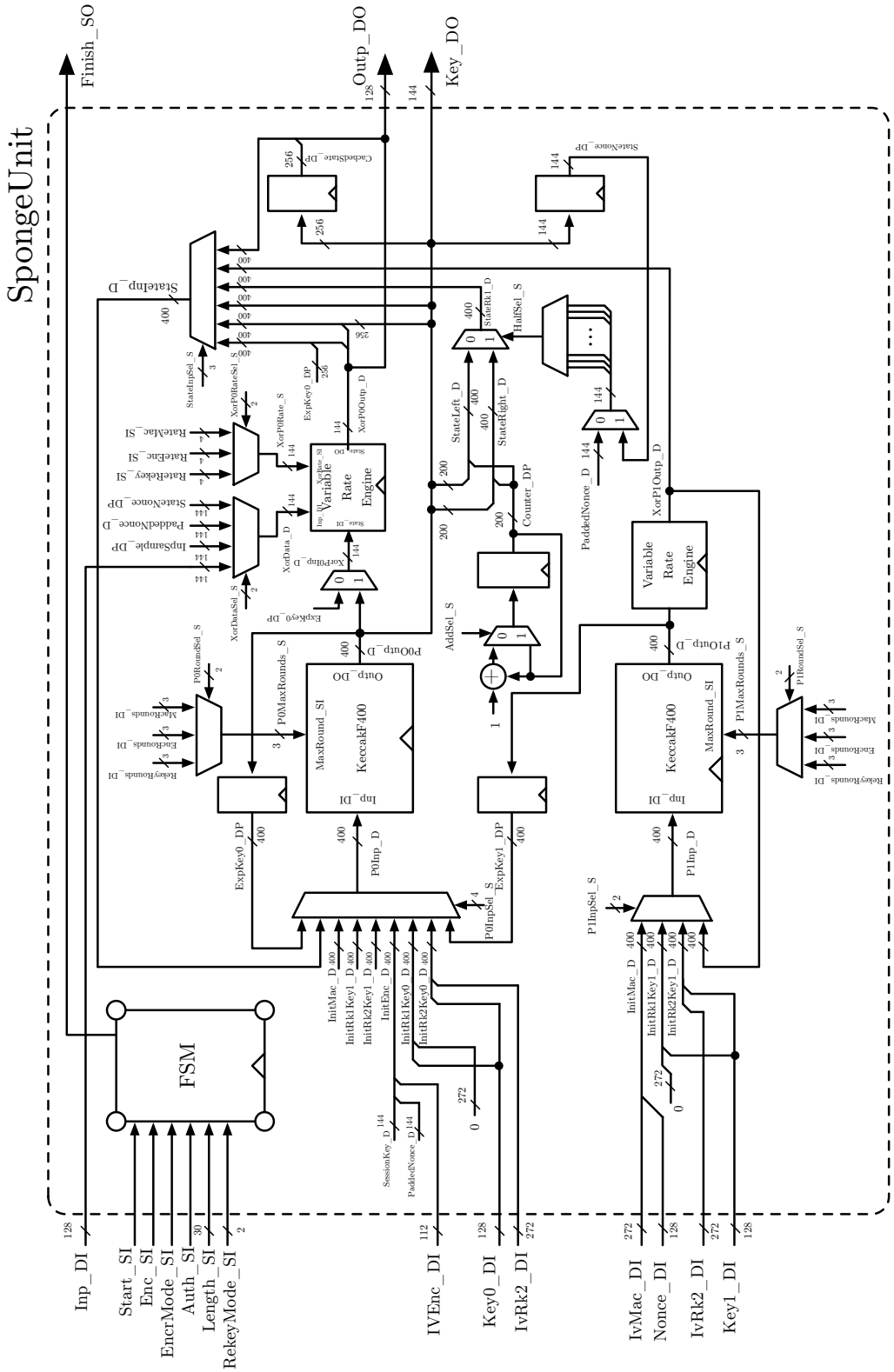



Figure 5.18: Architecture of the sponge unit.

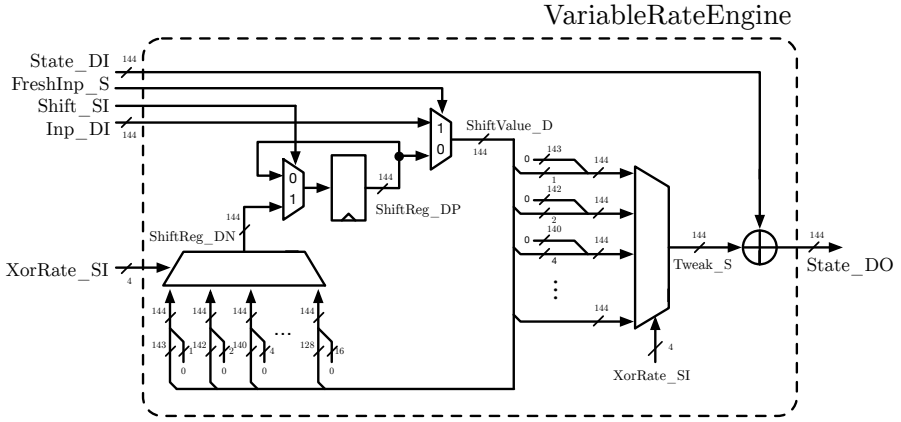



Figure 5.19: Architecture of the variable rate engine.

the typical-case process parameters against the worst-case parameters. One can clearly see the worst-case parameters are much slower compared to the typical-case parameters. This architecture fulfills the requirements to reach a clock period of 2.5 ns to achieve a maximum frequency of 400 Mhz. The iterative architecture is very flexible and allows a trade-off between the clock period and the throughput per cycle. Removing one round in the critical path decreases the clock period by about 33 %. Adding one additional round in the critical path increases the clock period by about 25 %.

## 5.10 TCDM Interfaces

The accelerator consists of two 32-bit TCDM interfaces for data transfer. Both interfaces are used in a single-duplex mode, which means one interface is only used for receiving data, and the other one is only used for transmitting data. Furthermore, both interfaces implement an interface converter between the 32-bit TCDM interface and the internally used 128-bit AXI-like interface. The TCDM interfaces are used to load and store data for encryption and decryption. Moreover, both interfaces are used to load and store the authentication tags, when using an encryption mode with authentication.

We now evaluate the bandwidth requirements of the TCDM interface to make a proper choice on the number of required TCDM ports of the accelerator. The bandwidth is determined by the throughput of the AES-based encryption algorithms, as well as the throughput of the permutation-based encryption algorithm. For both algorithms, we define a block size of 128-bit per encryption, although the permutation-based encryption mode supports smaller block sizes as well. Figure 5.22 shows the required TCDM bandwidth for different latencies of the encryption algorithms. The AES-based encryption algorithm requires five cycles to encrypt one block of data using the leakage-resilient mode of operation. According to Figure 5.22, this requires two TCDM ports in single-duplex operation to handle the bandwidth. Although the ECB- and XTS-mode are faster

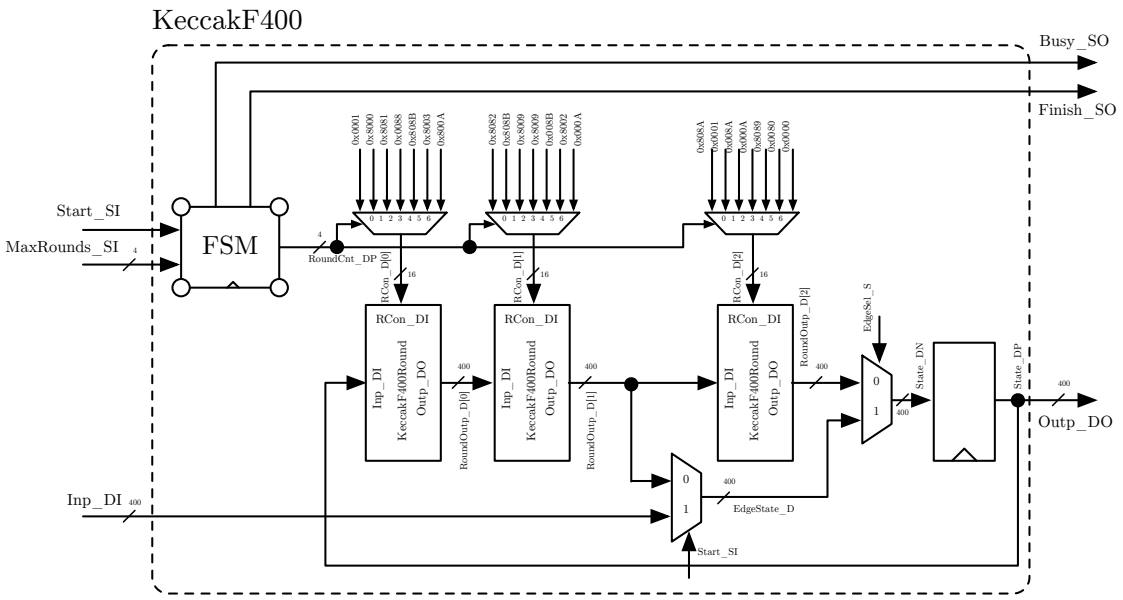



Figure 5.20: Architecture of the KECCAK-$f$[400] permutation function.

(they encrypt two words in five clock cycles rather than only one), the TCDM bandwidth is not optimized for those modes. The support for these modes of operation is only added for comparison reasons, but they are not the main target application of the accelerator.

## 5.11 Verification

In this section, we describe the verification approach used for *HWCrypt*. This covers the functional verification of the accelerator from a stand-alone hardware perspective, as well as the functional verification of the accelerator in the system perspective of the whole PULP chip.

### 5.11.1 Functional Verification

Functional verification is a big part of the design. Most of the hardware units contain their own randomized test bench. In fact, the development of *HWCrypt* followed a test-driven development approach. With the help of version control and continuous integration, the quality of the implementation was raised. Due to instant feedback of the continuous integration system, regressions were easily detected. The test suite for *HWCrypt* contains constraint randomized unit tests for the following design modules:

- *AES Dual Iterative*
- KECCAK-$f$[400]

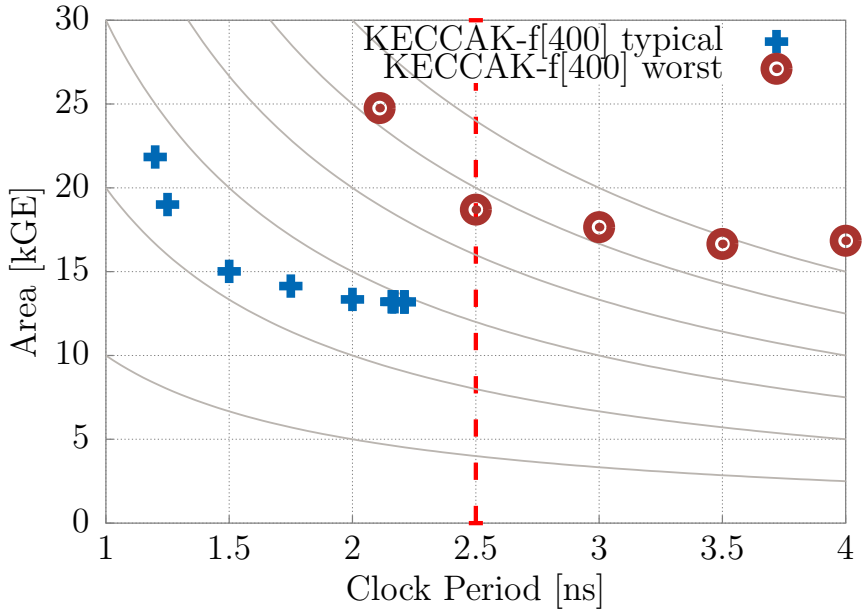



Figure 5.21: AT-plot of two KECCAK-$f$[400] architectures.

- AES unit
- Sponge unit
- LFSR
- TCDM receive interface
- TCDM transmit interface
- *HWCrypt*

All of these test benches follow the same approach. A `Driver` applies signals to the input interface, and a `Monitor` receives the response from the design under verification (DUV). The `Monitor` then compares the received response with the expected response, which is either computed directly in the test bench or read from a stimuli file. The stimuli file is generated from a golden model written in Python. A generic architecture of such a test bench design is shown in Figure 5.23.

Such a test bench design has similarities to the universal verification methodology (UVM) and is therefore called UVM-like test bench. However, we want to stress our test benches do not use any UVM-macros and are much simpler from the hierarchical point of view. Figure 5.24 shows the UVM-like test bench of the *HWCrypt* accelerator.

The architecture contains a `Driver` and a `Monitor` for both TCDM interfaces and a separate `Monitor` for the interrupt interface. Additionally, the test bench contains a so-called `Transactor` for the peripheral interface, which supports both driving and monitoring of the interface. All design blocks with a monitoring functionality (`Monitor` and `Transactor`) have a `Match` signal, which indicates whether the received response matches

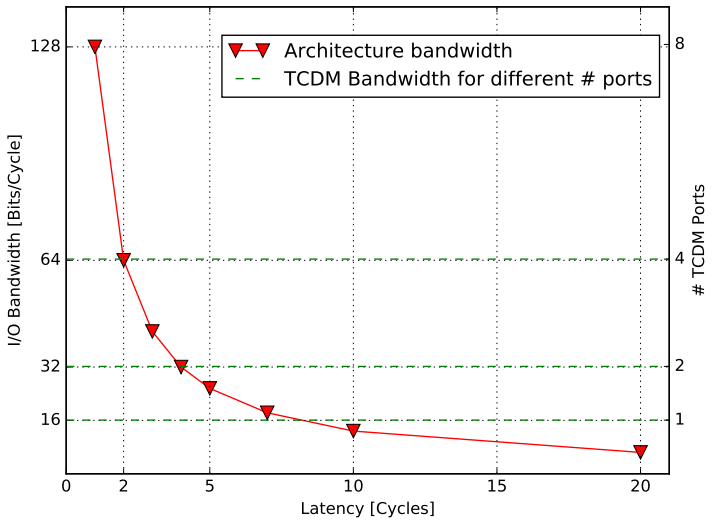



Figure 5.22: Bandwidth analysis for the TCDM interface.

the expected response of the golden model. If all `Match` signals are logic `1` for a test case, the test case passes. As soon as one `Match` signal is logic `0`, the test case fails.

### 5.11.2 Constraint Random Testing in Software

Constraint random testing is a state-of-the-art methodology in hardware development to reach a proper test coverage. In such a test architecture, the stimulies to test a hardware module are generated randomly but with certain constraints. One example for a constraint could be a configuration value for a mode of operation, which is only allowed to have certain values.

For testing the accelerator, we developed a golden model written in Python, which is capable of generating constraint random test cases. These test cases are used in the hardware test bench to test properly the accelerator. However, in this thesis we go one step further. We use the same golden model to convert the stimuli used in the hardware test bench into a C-program. This program is executed in the software test bench with the whole PULP system. To generate the C-program conveniently, we use *Jinja2* [53], a template engine for Python originally developed for the web development. The principle of a template engine is simple. First, we write a template of the C-program. This template is then filled with values coming from the golden model. For the use in *HWCrypt*, we write a template of the C-program, which either encrypts or decrypts data, performs a re-keying operation, or does a primitive operation. Then we generate a test case by using our golden model written in Python. The values of the generated test case are filled into the template by using *Jinja*. The C-program is then executed on the PULP system in a hardware-software co-simulation test bench.

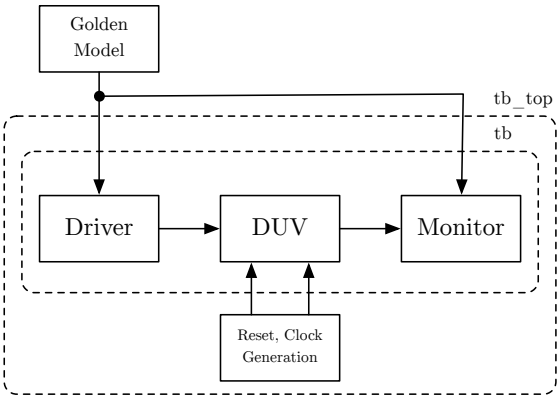



Figure 5.23: A generic test bench architecture.

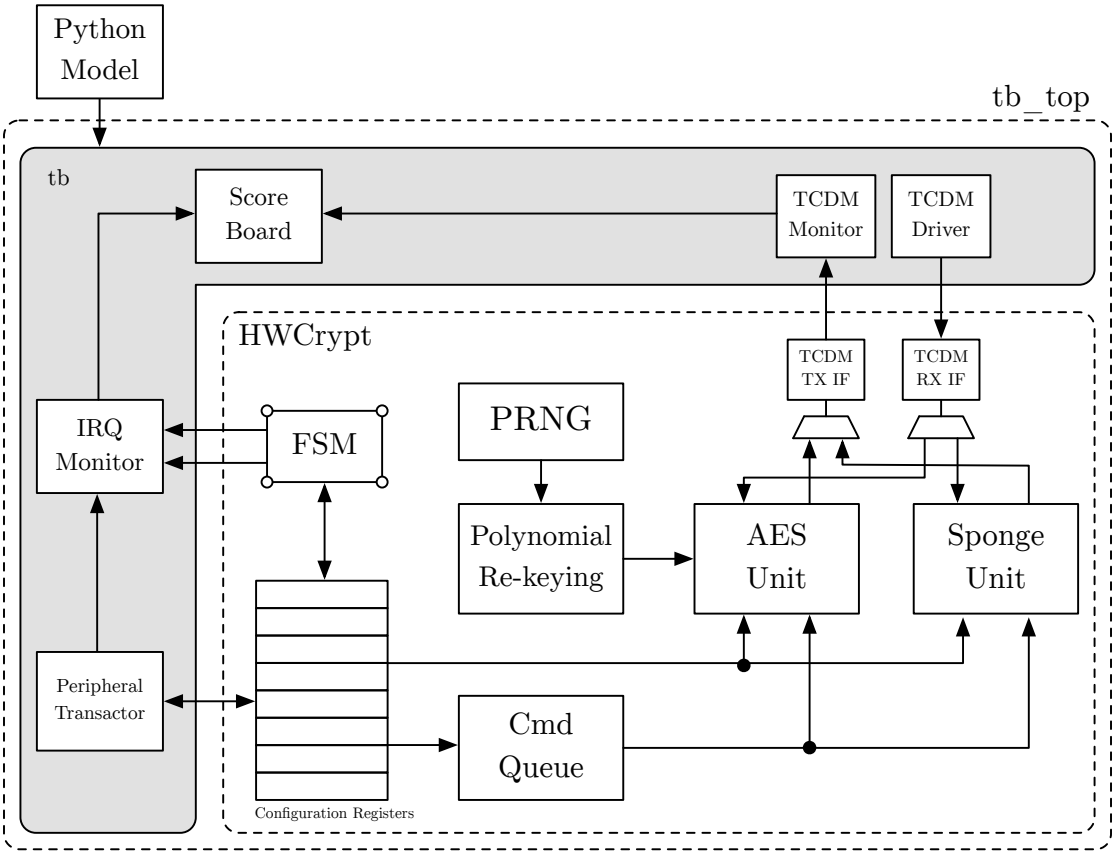



Figure 5.24: UVM-like test bench of *HWCrypt*.

Chapter 6

# Results

In this chapter, we evaluate the results of the hardware implementation of the cryptographic accelerator *HWCrypt*. Moreover, the software performance of certain parts of the accelerator is discussed. We show the throughput improvements achieved due to the hardware architecture. The overall efficiency of the accelerator in a real-life scenario is analyzed, and the improvements due to implementing a command queue are presented. Eventually, we discuss the backend results of the taped-out *Fulmine* ASIC.

## 6.1 Constraints

The *HWCrypt* accelerator is taped-out as part of a PULP chip using the UMC 65 nm technology. Previous tape-outs of PULP chips in this technology reach a frequency of 400 MHz when synthesizing with the worst-case process parameters. As presented in Chapter 5, the design of the cryptographic accelerator *HWCrypt* meets these timing requirements. The longest combinational path in *HWCrypt* is determined by the design of the AES-128 instance. As shown in Figure 5.15 the critical path of this design, a register- to-register (R2R) path, is given by 2.48 ns. The R2R path of the KECCAK-$f$[400] permutation defines the next group of critical paths with a clock period of 2.11 ns.

## 6.2 HWCrypt Accelerator

In Table 6.1, we present the synthesis results for *HWCrypt* using the worst-case process parameters ($V_{DD}$ = 1.08 V, T = 125 °C). This table also shows the relative utilization of each major component. The implementation of the peripheral interface including the register file of the accelerator is integrated in the top-level design of *HWCrypt*.

Table 6.1: Area breakdown of *HWCrypt* based on synthesis results.

| Component | Area [kGE] | Utilization [%] |
|---|---|---|
| *HWCrypt* | 349 | 100 |
| AES unit | 220 | 63.1 |
| Sponge unit | 61 | 17.4 |
| Polynomial Re-keying | 19.8 | 5.6 |
| CmdQueue | 11.6 | 3.3 |
| LFSR | 1.9 | 0.5 |
| TCDM RX interface | 2.3 | 0.6 |
| TCDM TX interface | 2.3 | 0.6 |

Table 6.1 indicates that with 63 % utilization the AES unit is the largest part of the design. As already discussed in Section 5.8, the size of the accelerator can be reduced when targeting only one encryption mode rather than acting as a universal evaluation platform. When only implementing the AES-based leakage-resilient mode, the sponge unit can be omitted. Furthermore, the AES unit can be improved significantly. Since the leakage-resilient mode only requires the encryption functionality of the block cipher, rather than both encryption and decryption, the area of the block cipher can be reduced significantly. Because the S-boxes for encryption and decryption mainly determine the size of the AES implementation, removing the decryption part roughly halves the area of the AES-128 instance. The decryption part of AES-128 in *HWCrypt* is only required for the ECB- and XTS-mode, which are added for evaluation purposes.

When only targeting the permutation-based sponge unit, we can omit the AES unit and the polynomial re-keying unit. This reduces the area of the design to less than 100 kGE. Furthermore, the sponge unit can be improved by using only one permutation instance to save about 20 % of the area. This would reduce the throughput for authentication when the encryption and authentication algorithm share the same configuration.

## 6.3 Advanced Encryption Standard

In this section, we discuss the performance results of different AES-128 implementations. We start with a software implementation and then present the results of the hardware-accelerated AES architecture using the primitive operations of the accelerator. Eventually, we discuss the performance of the AES-based modes of operation using the full hardware implementation of *HWCrypt*.

### 6.3.1 Software Implementation

Starting with an unoptimized software implementation of AES-128, we investigate the runtime on a PULP processor, which is shown in Table 6.2. To improve the performance of the vanilla implementation, all lookup tables are moved to the TCDM to get a single-cycle access to the table values in the best case. Furthermore, the addressing of the state data and loops are optimized. Table 6.2 shows the improved version has a speed-up of almost a factor of 10 compared to the unoptimized implementation. One encryption of a 128-bit word (including the key expansion algorithm) takes 2900 cycles. Further optimizations could exploit a T-Table approach, in which multiple operations are combined in lookup tables. This, however, increases the requirements on the TCDM. For a benchmark purpose, we encrypt 1 kB of data. In this experiment, we reach a performance of 162.8 cycles per byte (cpb). As indicated in [54], an assembler optimized implementation on an AVR microcontroller reaches a performance of 124.6 cpb.

Table 6.2: Evaluation results of software AES-128 implementation.

| Operation | Duration [cycles] |
|---|---|
| One encryption (unoptimized) | 27000 |
| One round (optimized) | 260 |
| One encryption (optimized) | 2900 |

### 6.3.2 Primitive Operation

For a performance evaluation, we implement the AES-128 encryption algorithm using the primitive operations provided by the accelerator. One primitive operation invokes one double-round function of AES. Therefore, this operation is invoked five times to reach the required ten rounds for AES-128. The expanded round-keys are computed in advance in software. The primitive operation supports encrypting or decrypting of two words in parallel. Encrypting two words takes 340 cycles with a warm instruction cache, but does not include computing the round-keys. Table 6.3 shows how the total runtime of 340 cycles on the PULP system is split. Benchmarking the encryption for 1 kB of data results in a performance of 10.6 cpb.

Table 6.3: Evaluation results of primitive-based AES-128 implementation.

| Operation | Duration [cycles] | Relative Duration [%] |
|---|---|---|
| Execution | 112 | 33 |
| Encryption | 35 | 10.3 |
| LSU stall | 162 | 47.6 |
| Load stall | 31 | 9.1 |
| Sum | 340 | 100 |

Table 6.3 indicates that the main duration is not spent on the actual encryption computation, but rather on the configuration and on the programming phase of the accelerator. More than 55 % are spent on stalling the processing core due to a TCDM contention (LSU stall) or due to a load stall. However, we mention that the primitive mode is not designed for performance. The primary goal of this operation mode is making the internal primitives accessible via the peripheral interface. This allows us to implement a hardware-accelerated implementation of any algorithm which relies on these primitives.

As shown in Table 6.3, the encryption step only takes 33 % of the overall time. A detailed analysis in the RTL simulation reveals that the actual encryption step only takes 35 cycles. Since five encryption steps are invoked, a single encryption step takes seven cycles. If we assume that writing one peripheral register takes only one clock cycle, the theoretical performance shown in Table 6.4 would be achieved.

Table 6.4: Theoretical primitive performance for AES-128.

| Operation | Duration [cycles] |
|---|---|
| State XOR | 8 |
| State transfer | 8 |
| Round-key transfer | 40 |
| Encryption | 35 |
| Sum | 91 |

Encrypting two 128-bits words would take 91 cycles, which equals a performance of 2.84 cpb. Although this shows a significant performance gain, the accelerator is still not optimized for the primitive operation. If the round-keys could be written in parallel to the encryption step, the performance still could be improved by a factor of about two.

### 6.3.3 Hardware Implementation

To evaluate the performance of the hardware implementation of AES-128, we use the ECB- and XTS-encryption mode. Compared to the software implementation from Section 6.3.1, the hardware architecture also includes the key expansion algorithm and the controlling part. Furthermore, we show the performance of the leakage-resilient mode of operation. For a benchmarking purpose, we encrypt 1 kB of data in all configurations. The ECB- and the XTS-mode use both cipher stages of the AES instances in parallel. Therefore, they would have in theory double the bandwidth compared with the leakage-resilient mode of operation, which only uses one. However, this bandwidth difference is not visible in Table 6.5 in which the ECB- and XTS-mode have a similar performance compared with the leakage-resilient mode. This is caused by the TCDM interface, which bandwidth is designed for the throughput of the leakage-resilient mode of operation. Disabling the masking functionality of the polynomial re-keying function would reduce the duration by 20 cycles.

Table 6.5: Evaluation results of a hardware implementation of AES-128 for 1 kB.

| Operation | Duration [cycles] |
|---|---|
| AES-128 ECB encryption | 395 |
| AES-128 XTS encryption | 405 |
| AES-128 leakage-resilient encryption[1] | 435 |

### 6.3.4 Summary

We now compare the presented results for the AES-based ECB encryption and leakage-resilient (LR) mode of operation with each other. Table 6.6 shows the performance results of different AES-based encryption algorithms. Clearly the hardware implementation of AES-128 shows a tremendous performance improvement compared to the optimized software implementation. The leakage-resilient mode of operation reaches a performance of 0.42 cpb including the polynomial re-keying with first-order masking. This performance value is comparable with cryptographic implementations on state-of-the-art Intel processors using the AES-NI instruction set extension. However, the AES-NI instruction set does not support any SCA countermeasures. As indicated in [37], parallelizable modes such as the ECB encryption reach a performance of 0.63 cpb on a modern Intel i7 *Skylake* CPU. For serial encryptions, the performance is reduced to about 2.65 cpb. This shows our hardware implementation of the ECB mode is already about twice as fast as an implementation on a modern Intel CPU. In theory, the ECB performance of the implemented AES-128 would be about 0.16 cpb, but the memory interface of *HWCrypt* is not designed for this bandwidth. The Intel implementation uses a pipelined version of the AES round function. When the pipeline is full, each clock cycle computes one round. To compute one AES-128 encryption it then takes 10 cycles, which results in a theoretical throughput of $\frac{10\,\text{cycles}}{16\,\text{byte}} = 0.63\,\text{cpb}$. Moreover, the leakage-resilient mode is a serial algorithm due to its data dependencies. This means the hardware implementation of *HWCrypt* outperforms an implementation of such an algorithm on an Intel CPU using AES-NI by about a factor of six.

Table 6.6: Performance comparison of AES-128 based algorithms for 1 kB.

| Operation | Duration [cycles] | Throughput [Gbit/s] | Performance [cpb] |
|---|---|---|---|
| AES-128 ECB software | 166700 | 0.0246 | 162.8 |
| AES-128 ECB primitive | 21280 | 0.154 | 10.6 |
| AES-128 ECB hardware | 395 | 8.3 | 0.38 |
| AES-128 LR hardware[1] | 435 | 7.5 | 0.42 |
| AES-NI ECB [37] | - | - | 0.63 |
| AES-NI serial [37] | - | - | 2.65 |

[1] Polynomial re-keying with 1-st order masking and shuffling of partial products.

## 6.4 Polynomial Re-keying Unit

This section evaluates the performance of the re-keying function in software and hardware. We start with the evaluation of the multiplication in the finite field GF($2^8$) and then continue with an unmasked and masked implementation of the polynomial multiplier.

### 6.4.1 Software Implementation

For comparison reasons, the polynomial re-keying function as defined in Section 4.3 is implemented in software on the PULP system. This re-keying function performs in total 256 multiplications in the finite field GF($2^8$) as its primary operation. For this reason, we first evaluate the performance of the GF($2^8$) multiplication in software.

#### 6.4.1.1 Multiplication in GF($2^8$)

This section evaluates different algorithms for the multiplication in GF($2^8$) in software on a PULP system. Algorithm 9 shows the iterative multiplication in the finite field GF($2^8$). The multiplication is performed using the irreducible Rijndael polynomial [7] $x^8 + x^4 + x^3 + x + 1$.

**Algorithm 9:** Iterative multiplication in GF($2^8$).

**Input**: $a, b \in \mathrm{GF}(2^8)$
**Output**: $c = a \cdot b \in \mathrm{GF}(2^8)$

```
1  p ← 0;
2  for c ← 0 to 7 do
3  |   if b & 0x01 then
4  |   |   p ← p ⊕ a;
5  |   end
6  |   a ← a << 1;
7  |   if a & 0x100 then
8  |   |   a ← a ⊕ 0x11b;
9  |   end
10 |   b ← b >> 1;
11 end
12 return p
```

A second algorithm to perform the multiplication in GF($2^8$) is using the exponential representation, which is shown in Algorithm 10. This algorithm uses table lookups to perform the multiplication, which can be implemented efficiently in software. However,

this algorithm requires 512-bytes of memory for the tables (256-bytes for the logarithm table and 256-bytes for the exponential table).

**Algorithm 10:** Multiplication in GF($2^8$) using exponential representation.

**Input**: $a, b \in \text{GF}(2^8)$
**Output**: $c = a \cdot b \in \text{GF}(2^8)$
1 **if** $a = 0$ **or** $b = 0$ **then**
2 | **return** 0
3 **end**
4 $tmp \leftarrow$ LOG_TABLE[$a$] + LOG_TABLE[$b$];
5 **return** EXP_TABLE[$tmp$];

Both algorithms are implemented as C-programs and are simulated on a PULP system. Algorithm 10 is implemented in two variants. First, the tables are stored in the L2-memory of PULP. Second, the tables are stored in the TCDM of the SoC. The naming of these functions is gf256mul_slow for Algorithm 9, gf256mul_fast, and gf256mul_fast_tcdm for Algorithm 10 respectively. Table 6.7 shows the simulated performance evaluation of the finite field multiplications running on the PULP system. The runtime is averaged for 256 invocations.

Table 6.7: Evaluation results for software implementations of a multiplication in GF($2^8$) averaged for 256 invocations.

| Operation | Duration [cycles] |
|---|---|
| gf256mul_slow | 132 |
| gf256mul_fast | 79 |
| gf256mul_fast_tcdm | 42 |

Based on the fast TCDM version of the multiplication in GF($2^8$), a polynomial multiplication is developed as described in Algorithm 5. Furthermore, the polynomial multiplication is extended to implement a masked polynomial multiplication as defined in Algorithm 8. For the software-based implementation, we use a masking order of three, which results in four polynomial multiplications in total. All measured values are averaged for 256 invocations.

Table 6.8: Evaluation results for a software implementation of a modular multiplication averaged for 256 invocations.

| Algorithm | Duration [cycles] |
|---|---|
| Polynomial multiplication | 10869 |
| Masked polynomial multiplication[2] | 41568 |

[2]Third-order masking

Table 6.8 shows the software evaluation results for the unmasked polynomial multiplication and masked multiplication using a masking order of three. Due to the iterative computation, the multiplication in software shows massive computation overhead in both evaluation cases. The slow software implementation argues this function to be implemented in hardware to increase the performance tremendously. Since PULP is a multi-core platform, the performance of the software implementation could be improved by exploiting the parallelism using multiple processing cores. Of course, this makes these processing cores unavailable for other computations.

### 6.4.2 Hardware Implementation

The software implementation highly indicates that a hardware implementation can perform much better. In fact, the hardware implementation computes the polynomial multiplication in 18 cycles which equals a speed-up by a factor of more than 500. Table 6.9 summarizes the performance of the hardware implementation of the unmasked and masked computation. Similar to Section 6.4.1, we use third-order masking for comparison reasons. The significant performance impact is not surprising since the polynomial multiplier is based on multipliers in GF($2^8$), which compute one multiplication within one clock cycle in hardware. This corresponds to a speed-up by a factor of about 40. Furthermore, the hardware architecture parallelizes the computation by using 16 multiplier stages. Although the performance of the software implementation is worse, it scales with architectural improvements in the hardware implementation.

Table 6.9: Evaluation results for a hardware implementation of a masked polynomial multiplication.

| Algorithm | Duration [cycles] |
|---|---|
| Polynomial multiplication | 18 |
| Masked polynomial multiplication[3] | 72 |

### 6.4.3 Random Bit Requirements

The masked polynomial multiplier requires random bits for the masks and the shuffling unit. In this section, we estimate the required number of bits for both operations.

[3] Third-order masking

#### 6.4.3.1 Masking

For $m$-th order masking, we require $m$ masks, each 128-bit wide. This results in the required random bits as defined in Equation 6.1.

$$\#\text{bits}_{\text{mask}} = m \cdot 128 \tag{6.1}$$

#### 6.4.3.2 Shuffling

The architecture of our shuffling unit supports different shuffling strategies. In this evaluation we consider the Fisher-Yates shuffling operation, which requires the most random bits. The Fisher-Yates shuffling algorithm can be reduced to the coupon collector's problem [55]. The expected value to draw all $N$ values is given in Equation 6.2.

$$E(X) = N \sum_{i=1}^{N} \frac{N}{N-i} \tag{6.2}$$

For certain cases, the modulus is used to reduce the random value to a certain interval as described in Section 4.3.1. Therefore, the expected value of draws for these cases is guaranteed to be 1. This improves the expected values of draws as shown in Equation 6.3.

$$E(X) = \sum_{i=0}^{N-1} \begin{cases} 1 & \text{if } i = \text{ 8,12,14,15} \\ \frac{N}{N-i} & \text{otherwise} \end{cases} \tag{6.3}$$

Computing the expected value of draws for $N = 16$ using Equation 6.3 results in $\lceil 28.09 \rceil = 29$ draws. Each draw equals a 4-bit random number from the PRNG. Given the expected number of draws, this result in $\#\text{bits}_{\text{shuffle,min}} = 116$-bits of required random data for one polynomial multiplication. Equation 6.4 combines the required bits for masking with a masking order of $m$ and shuffling, which performs $m + 1$ polynomial multiplications. First-order masking ($m = 1$) requires, at least, 360-bits of random data.

$$\begin{aligned} E\left(\#\text{bits}_{\text{single}}\right) &= \#\text{bits}_{\text{mask}} + E\left(\#\text{bits}_{\text{shuffle,min}}\right) \\ &= m \cdot 128 + (m+1) \cdot 116 \end{aligned} \tag{6.4}$$

At this point, only one draw per cycle is considered. Our hardware implementation evaluates multiple draws in parallel. Therefore, Equation 6.3 is extended for multiple

independent draws. In each cycle, $k$ stages evaluate an independent draw. Only one draw out of the $k$ stages is taken. All other draws are discarded. This results in Equation 6.5.

$$E(X) = \sum_{i=0}^{N-1} \begin{cases} 1 & \text{if } i = \text{ 8,12,14,15} \\ \frac{1}{1-\left(\frac{i}{N}\right)^k} & \text{otherwise} \end{cases} \tag{6.5}$$

For the shuffle strategy in the polynomial re-keying unit, the number of elements $N$ equals to 16. Equation 6.5 is evaluated for different numbers of parallel units $k$, which is shown in Figure 6.1. The more stages $k$ we have, the fewer draws are required.

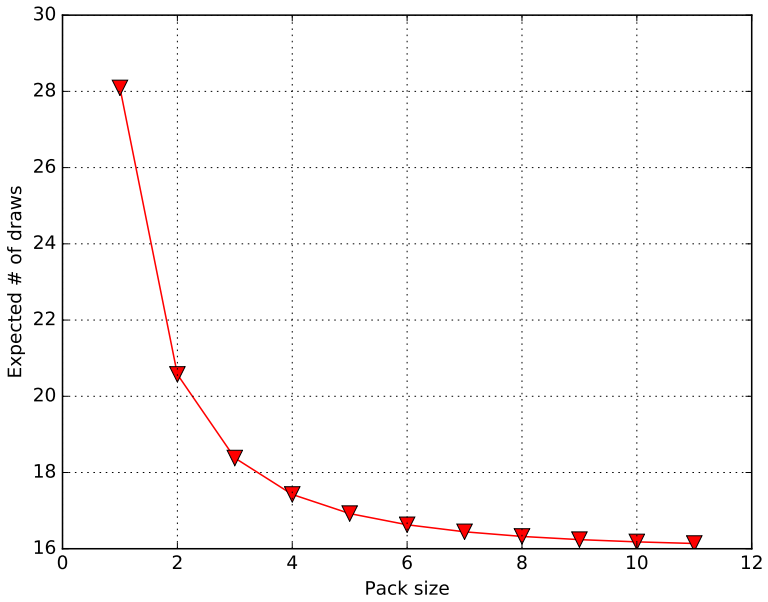



Figure 6.1: Parallelized coupon collector's problem.

The polynomial re-keying unit contains $k = 33$ evaluation stages and draws $N = 16$ values. This results in an expected value of $E\left(\#\text{bits}_{\text{shuffle,par}}\right) = \lceil 16.00106 \rceil = 17$ draws according to Equation 6.5. Each draw requires $33 \cdot 4$ bit of random data, which results in 132-bits of random data per draw. Given the expected value of number of draws, one can compute the required number of random bits. However, the first 4-bit value of the first draw guarantees to succeed to be a valid one. The remaining 128-bit of random data are not discarded. Instead, this data is reused for the first mask. This optimization leads to the required number of bits described in Equation 6.6. First-order masking ($m = 1$)

requires 4488-bit of random data in the architecture with 33 parallel evaluation stages.

$$\begin{aligned} E\left(\#\text{bits}\right) &= \#\text{bits}_{\text{mask}} + E\left(\#\text{bits}_{\text{shuffle,par}}\right) \\ &= (m+1) \cdot (17 \cdot 132) \end{aligned} \tag{6.6}$$

This random data requirement affects only the polynomial re-keying unit. This means the bit requirements are independent of the block size to be encrypted. Furthermore, the permutation-based re-keying functions do not require random data, which represents a major advantage of that architecture.

## 6.5 Performance Evaluation of the Sponge Unit

In this section, we evaluate the performance of the versatile hardware implementation of the sponge unit. The performance results of the permutation-based re-keying functions, as well as the implemented encryption mode, are presented. In total, this unit supports up to 20,000 different configuration options when considering the permutation-based encryption modes alone. This already shows the flexibility of this unit.

### 6.5.1 Re-keying Function

The sponge unit supports two different re-keying functions named *RK1* and *RK2*, which are described in Section 4.5.2. The performance of the hardware implementation in the PULP system of both re-keying functions is evaluated in this section.

#### 6.5.1.1 Re-keying Function RK1

The *RK1* re-keying function is based on a GGM-tree using the bits of the nonce. This operation is a bit-wise operation using each bit of the nonce, which is not configurable. However, the number of invoked rounds of the permutation function can be parametrized. In Table 6.10 we show the performance results for all possible round configuration. As described in Section 6.5.2, the measured values do not exactly reflect the actual time for computation. Due to unrelated instructions for starting and polling the accelerator, a bias of 57 cycles is measured. This needs to be taken into account for comparison. Table 6.10 shows the performance range of the *RK1* re-keying function from 345 cycles to 1209 cycles. Compared to the performance of polynomial re-keying, as shown in Table 6.9, the performance of *RK1* re-keying is worse. However, the construction of this re-keying function is DPA-safe by design. Additional countermeasures such as masking or hiding are not required. Furthermore, the permutation-based re-keying functions do not require random numbers.

Table 6.10: Performance results in cycles for *RK1* re-keying.

| Rounds | Duration [cycles] |
|---|---|
| 3 | 345 |
| 6 | 489 |
| 9 | 533 |
| 12 | 777 |
| 15 | 921 |
| 18 | 1065 |
| 20 | 1209 |

#### 6.5.1.2 Re-keying Function RK2

The permutation-based re-keying function *RK2* supports two different configuration options. The number of invoked rounds for the permutation function and the data rate on which the nonce is processed are parametrizable. Those two options span a matrix of different configurations as shown in Table 6.11. Compared to the *RK1* re-keying function, this facilitates more configuration options.

Table 6.11: Performance results in cycles for *RK2* re-keying.

| Rate \ Rounds | 20 | 18 | 15 | 12 | 9 | 6 | 3 |
|---|---|---|---|---|---|---|---|
| 1-bit | 1209 | 1065 | 921 | 758 | 565 | 489 | 345 |
| 2-bit | 633 | 561 | 489 | 417 | 345 | 273 | 201 |
| 4-bit | 345 | 309 | 273 | 237 | 201 | 165 | 129 |
| 8-bit | 201 | 183 | 165 | 147 | 129 | 111 | 93 |

### 6.5.2 Encryption Mode

We also evaluate the real performance of the permutation-based encryption mode on the PULP system. This mode of operation supports two degrees of freedom for configuration, namely the number of invoked rounds of the permutation function and the rate of the processed data. This allows us to trade-off between the performance and the security of the encryption mode. Table 6.12 shows the performance results for encrypting 1 kB of data using different configurations of the sponge unit. These performance merits do not contain any overhead due to re-keying. All values are measured on the PULP system in software and, therefore, reflect the real performance numbers including overhead for starting and polling the accelerator in software. As visualized in Table 6.12 the sponge unit supports various configuration options. When considering the performance results for a high throughput, the encryption duration in cycles is similar for different configurations. For these cases, the encryption bandwidth is limited by the bandwidth of the

TCDM interface.

Table 6.12: Performance results in cycles for permutation-based encryption.

| Rate \ Rounds | 20 | 18 | 15 | 12 | 9 | 6 | 3 |
|---|---|---|---|---|---|---|---|
| 1-bit | 65596 | 57406 | 49216 | 41020 | 32830 | 24640 | 16450 |
| 2-bit | 32830 | 28735 | 24640 | 20545 | 16450 | 12335 | 8257 |
| 4-bit | 16450 | 14398 | 12355 | 10303 | 8251 | 6208 | 4162 |
| 8-bit | 8251 | 7234 | 6195 | 5182 | 4156 | 3139 | 2100 |
| 16-bit | 4156 | 3643 | 3117 | 2626 | 2113 | 1600 | 1087 |
| 32-bit | 2113 | 1852 | 1587 | 1339 | 1087 | 835 | 580 |
| 64-bit | 1087 | 961 | 813 | 700 | 574 | 448 | 445 |
| 128-bit | 574 | 511 | 435 | 448 | 448 | 448 | 444 |

These timings are measured in software and, therefore, contain additional overhead due to unrelated instructions for starting and polling the accelerator. We analyzed the RTL simulation and measured an average overhead of 57 cycles, which can be subtracted from the performance results of Table 6.12 to yield the raw encryption performance of the accelerator. This results in a throughput range between 50 Mbit/s and 8.47 Gbit/s when using a clock frequency of 400 MHz. This broad range of configuration shows the flexibility of the permutation-based encryption mode again.

## 6.6 Accelerator Efficiency

Having a fast encryption core is only one side. If there is much overhead due to the communication interface, the software cannot utilize the accelerator enough to reach a good efficiency. Therefore, the performance of the communication interface is analyzed. For this experiment, we consider decrypting 8 kB of data from the L2-memory of the PULP system. The experiment uses the AES-based leakage-resilient mode of operation with first-order masking. The data is loaded in 1 kB blocks via the DMA controller from the L2-memory into the TCDM. The DMA controller generates an interrupt and triggers the cryptographic accelerator to start a fresh decryption with the newly available data. This is repeated until all eight blocks are decrypted. We aim for an efficiency above 90 % in a practical scenario.

To measure the efficiency, we take the following three phases of the accelerator into account:

(A) Decrypting one 1 kB block of data

(B) Initial configuration of the accelerator

(C) Reconfiguration including the interrupt latency

Given these values, the efficiency of the accelerator is computed as follows.

$$\text{Efficiency} = \frac{A}{A+B+C} \tag{6.7}$$

Since the initial configuration $B$ is only done once, decrypting multiple blocks of data will gain a better efficiency.

### 6.6.1 Without a Command Queue

First, the efficiency is evaluated without having a command queue. Because we only have one context, the configuration- and interrupt-latency impact the overall performance of the accelerator. We first evaluate the efficiency of a single decryption operation. The efficiency of a single operation depends on whether it is the first operation or a later one. The first operation suffers from the initial configuration latency and cache misses. In addition to the interrupt latency, the second operation suffers from cache misses in the interrupt handler. Later decryptions are faster since cache misses do not influence the efficiency as much as in the first two operations. Moreover, the block size also influences the efficiency. The larger the block size, the more efficient the operation. The efficiency is improved since there are fewer context switches due to an increased block size. This behavior is visualized in Figure 6.2. This figure shows the first, second, and later decryptions for different block sizes. However, using a large block size of 16 kB is not practical in the PULP environment with a limited TCDM.

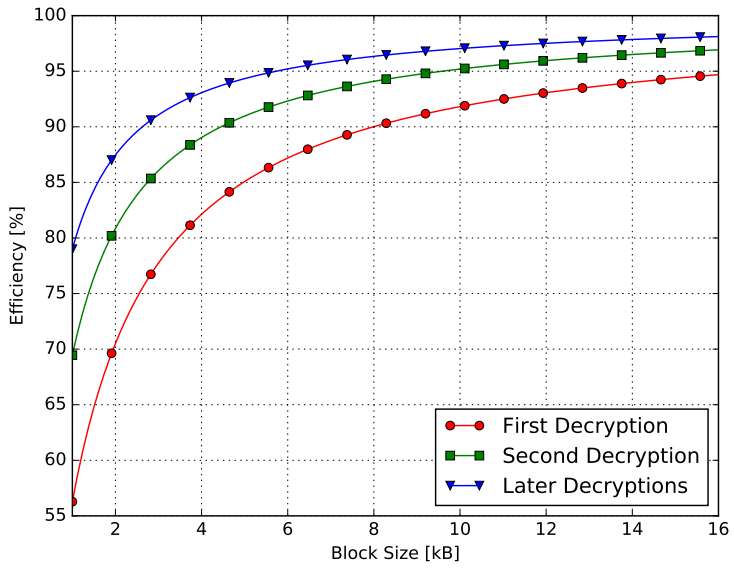



Figure 6.2: Efficiency for increasing block size.

In Figure 6.3 we show the total efficiency in a streaming application. The experiment decrypts 8 kB using different block sizes. When using a block size of 1 kB, the efficiency does not raise much above 75 %. By taking larger block sizes, the efficiency can be increased. However, with a reasonable block size and a reasonable size of data to be decrypted, 90 % efficiency is not reached in practice.

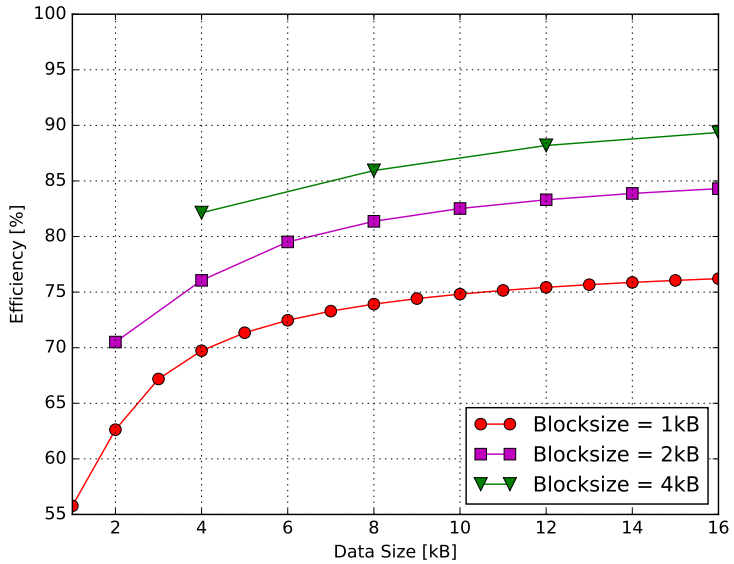



Figure 6.3: Total efficiency for different block sizes.

### 6.6.2 With a Command Queue

To enhance the efficiency, the architecture of *HWCrypt* contains a command queue to deal with multiple contexts. The software can configure the next operation while the accelerator is still busy with its current job. This mitigates the performance loss due to the long interrupt latency. The data source – in our case the DMA controller – can immediately configure the next operation when a DMA transfer completes.

In Figure 6.4 we show the total efficiency for the same experiment as done in Section 6.6.1. The result shows the efficiency of the accelerator reaches the 90 % mark for a block size of 1 kB when decrypting about 7 kB of data, which equals seven blocks. In a streaming application, this is a reasonable scenario. Using the command queue, we utilize fully the accelerator. This means increasing the block size as it is done in the previous experiment would not increase the performance. The efficiency of the architecture with a command queue now only depends on the initial configuration latency.

This evaluation shows a command queue, or a similar interface to handle multiple contexts, is an essential feature to fully utilize the accelerator. Without such an optimized

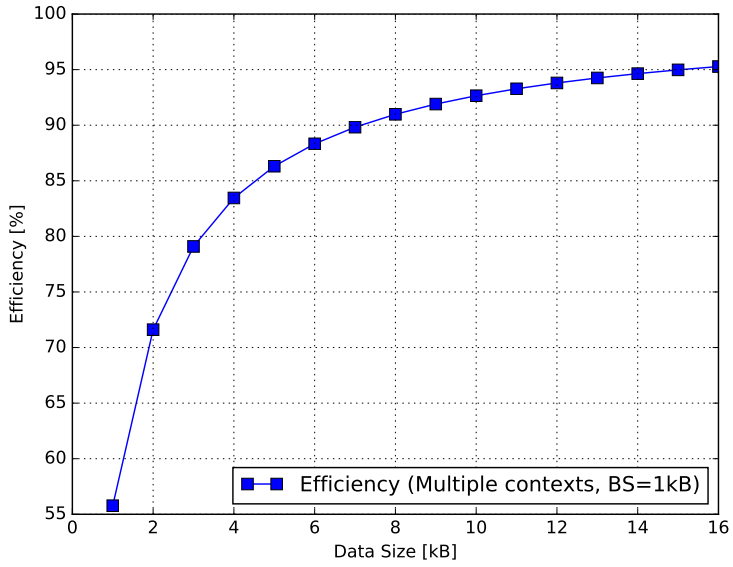



Figure 6.4: Practical accelerator efficiency with a command queue.

configuration interface, the configuration time and the interrupt latency are too high to be efficient in practice. Such interfaces are also used in commercial products, which often support a double-buffer [35] interface. This needs to be taken into account when developing a new accelerator.

## 6.7 Fulmine - ASIC

The *HWCrypt* accelerator is taped-out in the *Fulmine* ASIC. This chip contains a four-core PULP system with two[4] accelerators inside the cluster. Moreover, the OpenRISC core is modified to supported an extended instruction set with optimizations for signal processing operations. In Table 6.13 we summarize the main features of *Fulmine*. As indicated, *Fulmine* supports two different maximum frequencies. This is a result of the backend design. A multi-mode multi-corner (MMMC) strategy was used to have different operating modes. The clock speed of the cryptographic accelerator was reduced to meet the area requirements for the chip. Figure 6.5 shows the final layout of the *Fulmine* ASIC.

[4] *Fulmine* consists a second accelerator for convolutional operations, which is unrelated to this project

Table 6.13: *Fulmine* features.

| | |
|---|---|
| Technology | UMC 65 nm |
| Package | QFN64 |
| Dimensions | 2626 $\mu$m x 2626 $\mu$m |
| Processing Cores | 4 |
| Instruction set | OpenRISC + custom instructions |
| Accelerator | Convolutional and cryptographic accelerator |
| Max. frequency[5] [MHz] | 285 |
| Max. *HWCrypt* frequency[5][MHz] | 250 |
| Max. frequency[6] [MHz] | 400 |
| L2-memory [kB] | 192 |
| TCDM [kB] | 64 |
| Total area [kGE] | 2500 |
| Cluster area [kGE] | 1660 |
| *HWCrypt* area [kGE] | 340 |

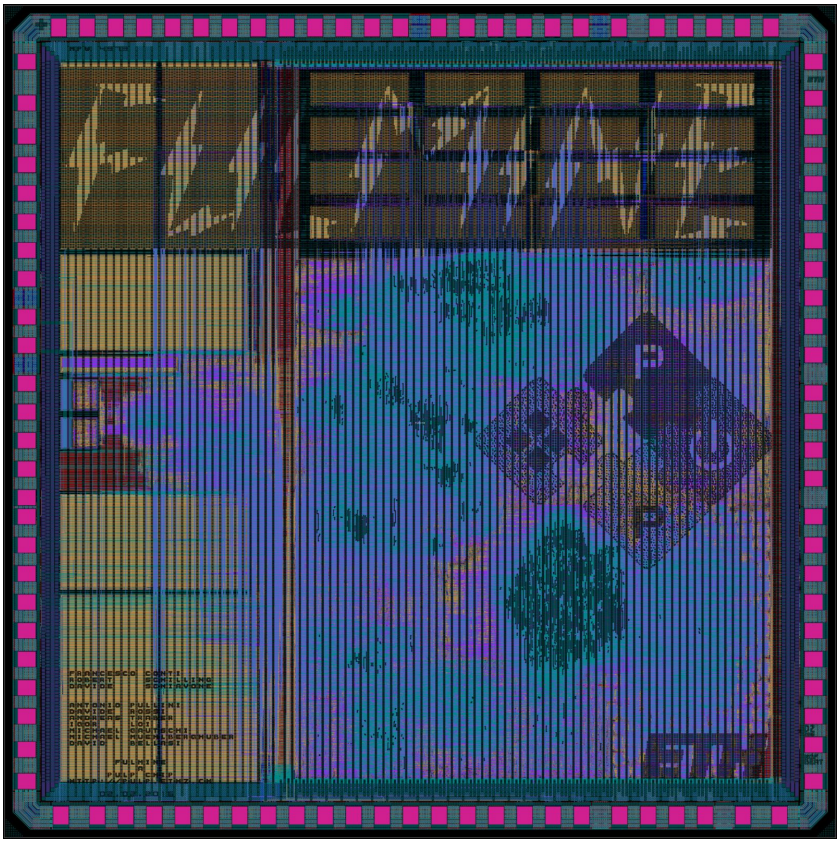


Figure 6.5: Layout of the *Fulmine* ASIC.

[5] $V_{DD}$ = 1.08 V, T = 125 °C
[6] $V_{DD}$ = 1.2 V, T = 25 °C

Chapter 7

# Conclusion and Future Work

In this thesis, we present the first ASIC implementation of a leakage-resilient cryptographic mode based on polynomial re-keying and a 2PRG construction. Moreover, we also show the first ASIC design of PASEC, a permutation-based leakage-resilient authenticated encryption scheme. Both encryption modes are designed to withstand DPA attacks and rely on the concept of fresh re-keying. This ensures that a session key is used for only one single encryption which mitigates DPA attacks on the cipher by design. Instead, it shifts the requirements for DPA safety from the cipher to the re-keying function. To avoid a DPA attack on the polynomial re-keying unit, we implement masking and hiding as DPA countermeasures. In detail, the architecture supports masking up to order 255 and multiple shuffling strategies to implement hiding. The permutation-based re-keying functions are DPA-safe by design and do not require any additional countermeasures. Due to their flexible design, these re-keying functions support a trade-off between security and throughput during runtime. Moreover, we also implement a permutation-based encryption mode with a fine-grained configuration matrix. Apart from the encryption mode, PASEC also defines a leakage-resilient authentication mode. This feature provides integrity of the ciphertext. The authentication algorithm detects any tampering on either the nonce, on the authentication tag, or on the ciphertext.

We implement the presented cryptographic modes in a hardware accelerator named *HWCrypt* as part of the cluster of a PULP system. This PULP system is taped-out using the 65 nm technology of UMC on the *Fulmine* ASIC. *Fulmine* is a four-core PULP architecture containing two hardware accelerators and improved OpenRISC processing cores. The cryptographic accelerator *HWCrypt* is designed to support protecting the following two scenarios. First, *HWCrypt* supports encrypting the TCDM. The encrypted data is finally stored in the L2-memory of the SoC to protect the content. Second, the encrypted data of the TCDM may be sent over an I/O interface of the chip. *HWCrypt* ensures that transmitted data is encrypted and ensures safety against side-channel attacks.

The accelerator provides a highly-flexible evaluation platform for leakage-resilient encryption algorithms based on fresh re-keying. Due to the flexibility of the architecture, millions of different configurations are possible. This allows us to evaluate the security of the implemented modes of operation in a fine-grained way.

## 7.1 Future Work

The cryptographic accelerator *HWCrypt* is the first step towards evaluating the security of leakage-resilient encryption modes for high-performance processing architectures. Since the primary goal of this work is the implementation of a flexible evaluation platform, future work may integrate the evaluated algorithms into concrete products. This may result in a secure DMA controller, which performs encryption and authentication of different, configurable memory ranges transparently. Moreover, the DMA controller may be capable of dealing with I/O peripherals such as SPI. This enhanced architecture would support both the memory encryption scenario, as well as the secure communication scenario.

Using the leakage-resilient modes of *HWCrypt*, the efficiency of the accelerator scales with the amount of data to be processed. Due to the re-keying function, which is performed before the encryption part, the performance of small encryptions is decreased. Future work might improve the performance of *HWCrypt* for very small block sizes. This would make the accelerator more suitable for small communications used in the context of RFID.

The design and implementation of the cryptographic accelerator *HWCrypt* are only the initial steps to evaluate leakage-resilient primitives for IoT. To verify the security of the leakage-resilient primitives, we will develop a proper DPA measurement setup for the PULP chip. This setup contains the external host for booting the PULP chip, as well as a possibility to accurately measure the power consumption of *Fulmine*. This evaluation platform can be reused for future measurements of PULP chips using the same package and bonding layout. To state the security, we will try to attack the *Fulmine* chip with the cryptographic accelerator. The results will reveal whether the encryption modes with its countermeasures can withstand sophisticated DPA attacks. This evaluation of the ASIC will eventually lead to safer cryptographic primitives used in the IoT.

Appendix A

# HWCrypt Accelerator Datasheet

In the appendix, we describe all operating modes of the *HWCrypt* cryptographic accelerator, which is part of the *Fulmine* ASIC. *Fulmine* is a four-core PULP chip fabricated in the UMC 65 nm technology with two hardware accelerators. It consists of *HWPE*, a convolutional accelerator and *HWCrypt*, a cryptographic accelerator. The datasheet in the appendix aims to give an understanding of the supported features of *HWCrypt* and how to use them.

## A.1 Features

- Leakage-resilient memory encryption based on AES-128
- Leakage-resilient memory encryption based on a KECCAK-$f$[400] permutation function with authentication
- AES-128 XTS memory encryption
- AES-128 ECB memory encryption
- Leakage-resilient hardware-accelerated re-keying functions based on a polynomial multiplication and on the KECCAK-$f$[400] permutation with countermeasures against DPA attacks.
- Hardware accelerated AES round function and KECCAK-$f$[400] permutation

## A.2 Applications

*HWCrypt* is a cryptographic accelerator for evaluating state-of-the-art leakage-resilient cryptographic algorithms for the next generation Internet-of-Things processors. This accelerator is used to protect the cluster of a PULP system against side-channel attacks by allowing the software to make encrypted transfers beyond the cluster boundary. With this protection both the memory transfers to the L2-memory and the communication transfers, are secured from adversaries. Furthermore, the protection contains countermeasures against side-channel attacks.

## A.3 Description

Figure A.1 depicts the functional block diagram of *HWCrypt*. It shows the major components of the accelerator, which are described below.

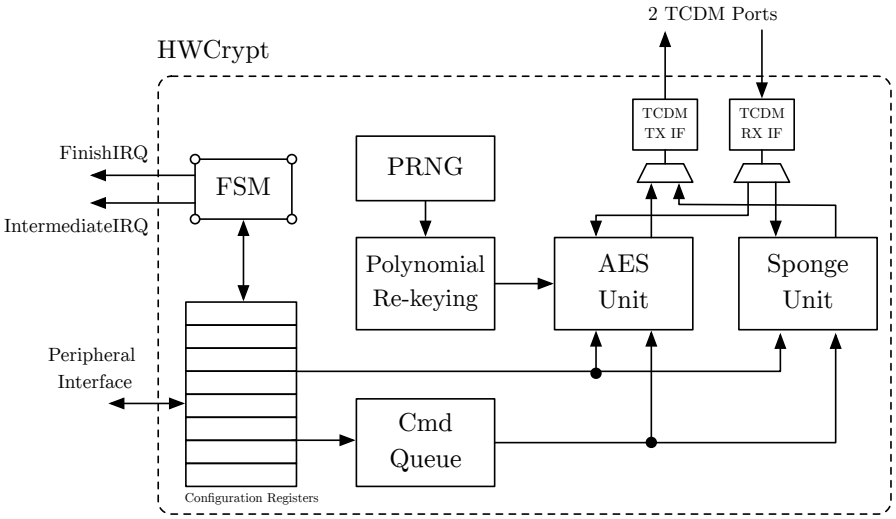



Figure A.1: Functional block diagram

**Peripheral Interface** — A 32-bit configuration interface, which is connected to the cluster-internal peripheral interconnect. This interface allows the processing cores to configure, start, and monitor the accelerator.

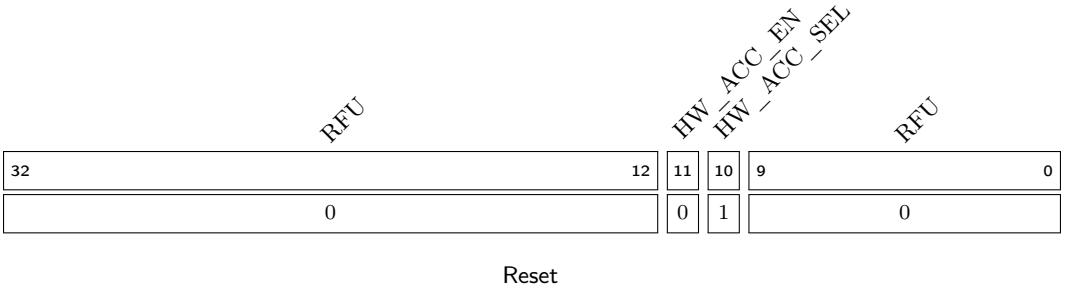
**TCDM Interface** — Two 32-bit TCDM interfaces are used to read and write data from the TCDM. The interfaces are also used to read and write the authentication tags.

| | |
|---|---|
| **Polynomial Re-keying Unit** | The polynomial re-keying unit implements a re-keying function based on a polynomial multiplication. This unit contains masking and shuffling to protect the algorithm against DPA attacks. |
| **Cmd Queue** | A command queue supports multiple-context computation for pending operations in a first-in-first-out scheme. |
| **PRNG** | A pseudo random number generator based on a 132-bit linear feedback shift-register provides random data for masking and shuffling. |
| **AES Unit** | The AES unit provides support for AES-128 ECB, AES-128 XTS, and the leakage-resilient mode of operation. Furthermore, this unit supports post-processing of session keys to avoid them to be invertible. |
| **Sponge Unit** | The sponge unit supports two permutation-based re-keying functions, an encryption mode, and authentication of the ciphertext. |

## A.4 *Fulmine* Configuration

The *Fulmine* ASIC supports two different accelerators, namely one accelerator for convolutional operations *HWPE* and one for cryptographic operations *HWCrypt*. The usage of both accelerators is mutually exclusive, which means one can only use one accelerator at a time. The reason behind this is that there is a limited number of TCDM ports of the internal interconnect. To use the right accelerator, the internal TCDM port multiplexer needs to be configured. Furthermore, *Fulmine* supports clock-gating the selected accelerator. Both configurations are performed in the `CLUSTER_CTRL_REG` register as defined in Register A.1.

Register A.1: CLUSTER_CTRL_REG. (0x10200018)




**HW_ACC_SEL** Selects the used accelerator. Logic `0` selects *HWCrypt*; logic `1` selects *HWPE*. The default selection while startup is *HWPE*.

**HW_ACC_EN** Logic `1` enables the clock signal of the accelerator. Only the selected accelerator by the `HW_ACC_SEL` bit is clocked.

## A.5 Interface Description

### A.5.1 Peripheral Interface

Table A.1 describes the signal interface of the *HWCrypt* accelerator. Apart from the clock and reset signals, the accelerator has three different interfaces. First, it contains a peripheral interface. This interface is connected to the cluster-internal peripheral interconnect. Processing cores can perform a register configuration of the accelerator via this interface. Furthermore, the peripheral interface of *HWCrypt* allows the processing cores to monitor the operation by reading the status registers. Second, the accelerator provides two 32-bit TCDM interfaces for data access. These interfaces are used by the accelerator to read and write the processed data from the TCDM. Moreover, *HWCrypt* uses these interfaces to read and write the authentication tags. Third, the accelerator contains two interrupt- or event-lines to notify the processing cores that an intermediate job or the last job has finished.

Table A.1: *HWCrypt* signal interface.

| Signal | Direction | Description |
|---|---|---|
| `Clk_CI` | **Input** | Clock input for the accelerator. |
| `Rst_RBI` | **Input** | Reset signal. Active low. |
| `FinishIrqEvent_SO` | **Output** | Finish interrupt line. Active high. |
| `IntermediateIrqEvent_SO` | **Output** | Intermediate interrupt line. Active high. |
| `PeriphReq_SI` | **Input** | Request signal to the peripheral interface. |
| `PeriphAddr_DI[31:0]` | **Input** | Address of the peripheral interface to be accessed. |
| `PeriphWen_SI` | **Input** | Write enable signal for peripheral interface. Active low. |
| `PeriphWdata_DI[31:0]` | **Input** | Data to be written to the accelerator. |
| `PeriphBe_SI[3:0]` | **Input** | Byte enable signal. Implementation ignores this signal. All accesses are treated as 32-bit accesses. |
| `PeriphId_SI[4:0]` | **Input** | ID of the request. |
| `PeriphGnt_SO` | **Output** | Grant signal for peripheral accesses. Read accesses are always granted. Write accesses are only granted if the accelerator is not busy or if the write access is referred to a queue register and the command queue is not full. |
| `PeriphRvalid_SO` | **Output** | Valid data signal. Goes high one cycle after the corresponding grant signal. |
| `PeriphRdata_DO[31:0]` | **Output** | Data read from the accelerator. |
| `PeriphId_SO[4:0]` | **Output** | ID of response. Must match the ID of the request. |
| `PeriphRpc_SO` | **Output** | Error code. Always `0`. |
| `TCDMRxReq_SO` | **Output** | Data request for the receiving TCDM interface. Active high. |
| `TCDMRxAddr_DO[31:0]` | **Output** | TCDM address of data to be read. |
| `TCDMRxWen_SO` | **Output** | Write enable signal for receiving TCDM interface. Always `1` for receiving operations. |
| `TCDMRxWdata_DO[31:0]` | **Output** | Always `0` since write operations are not used. |
| `TCDMRxBe_SO[3:0]` | **Output** | Byte enable signal. Only 32-bit transfers supported. Always set to `1111`. |
| `TCDMRxGnt_SI` | **Input** | Grant signal when a transfer is valid. Active high. |
| `TCDMRxRvalid_SI` | **Input** | Valid data signal. Goes high one cycle after the corresponding grant signal. |
| `TCDMRxRdata_DI[31:0]` | **Input** | Data response from the TCDM. |

Table A.2: *HWCrypt* signal interface continued.

| | | |
|---|---|---|
| `TCDMTxReq_SO` | **Output** | Data request for the transmitting TCDM interface. Active high. |
| `TCDMTxAddr_DO[31:0]` | **Output** | TCDM address of data to be written. |
| `TCDMTxWen_SO` | **Output** | Write enable signal for transmitting TCDM interface. Always `0` for transmitting operations. |
| `TCDMTxWdata_DO[31:0]` | **Output** | Data to be written to the TCDM. |
| `TCDMTxBe_SO[3:0]` | **Output** | Byte enable signal. Only 32-bit transfers supported. Always set to `1111`. |
| `TCDMTxGnt_SI` | **Input** | Grant signal when a transfer is valid. Active high. |
| `TCDMTxRvalid_SI` | **Input** | Valid data signal. Goes high one cycle after the corresponding grant signal. |
| `TCDMTxRdata_DI[31:0]` | **Input** | Transmitting interface does not use the read data signal. |

### A.5.2 Interrupt Interface

*HWCrypt* supports two different interrupts or events. The intermediate interrupt is raised if a job has finished while there are still pending jobs in the command queue. A finished interrupt is raised if all pending jobs have finished. Figure A.2 shows the interrupt behavior of the accelerator with three jobs. The first two finished jobs raise `IntermediateIrqEvent_SO`. The last operation raises the `FinishIrqEvent_SO` interrupt.

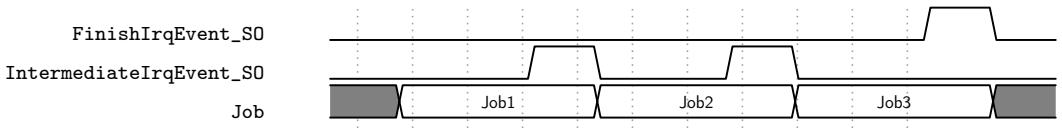



Figure A.2: Interrupt handling of *HWCrypt*.

### A.5.3 TCDM Interface

In Figure A.3 and in Figure A.4, we show the main communication protocol for the TCDM interface. First, we show a read transaction. The master controller of the interface performs a read request by pulling the `Req_SO` and the inverted `Wen_SO` signal to logic high together with the memory address to be read. The slave then grants the access by signalling the `Gnt_SO` with logic high. One cycle after the grant signal the `Rvalid_SI` is set to high by the slave together with the memory data on the `Rdata_DI` lines. The

grant signal may already be raised in the same clock cycle to logic high when the request is performed.

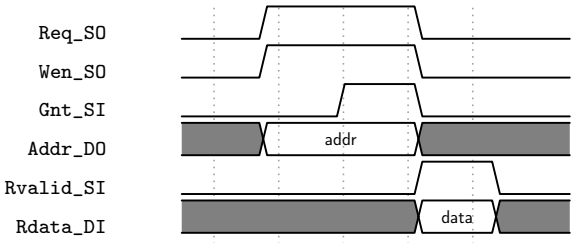



Figure A.3: TCDM read transaction.

The write transaction operates similarly as the read transaction as depicted in Figure A.4. The request is performed by pulling the `Req_SO` to logic high with the `Wen_SO` signal staying at logic zero. At the same time the target address `Addr_DO`, and `Wdata_DO` are set to the values for the request. When the TCDM memory controller raises the grant signal `Gnt_SI`, the transaction to the memory is successful. The grant may be raised in the same cycle as the request is performed

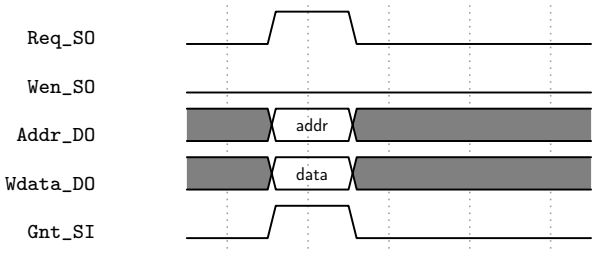



Figure A.4: TCDM write transaction.

## A.6 Register Map

Table A.3 shows the register map of *HWCrypt*. All registers are 32-bit wide and are accessible by the processing core via the peripheral interface. As indicated in Table A.3, registers such as `KEY0`, which are larger than 32-bit, are defined by an address range. Sub-registers are then named with a numbered postfix. The register map is visible from the processing cores at a base address of `0x10201000`.

Table A.3: Register map of *HWCrypt*.

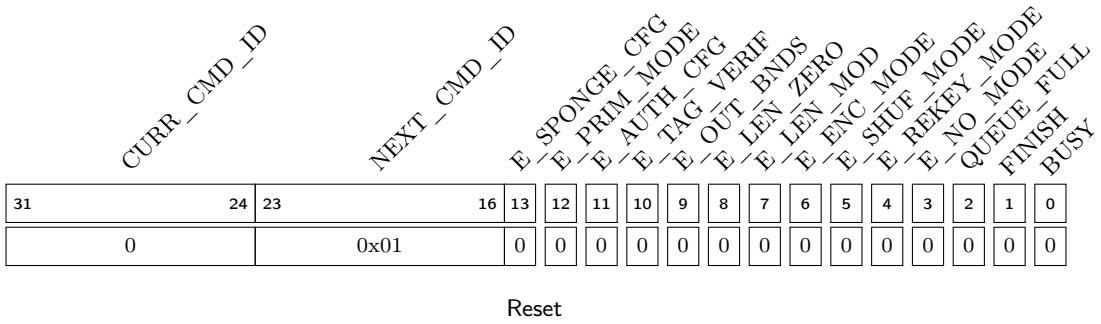
| Address | Name | Access | Description |
|---|---|---|---|
| 0x00 | `HWSTAT` | R | Status register to monitor the current state of *HWCrypt*. |
| 0x04 | `HWCTRL` | W | Control register to configure and start a new operation. |
| 0x08 | `TCDM_SRC` | R/W | TCDM data source address. |
| 0x0C | `TCDM_DST` | R/W | TCDM data destination address. |
| 0x10 | `TCDM_TSRC` | R/W | Tag source address. |
| 0x14 | `TCDM_TDST` | R/W | Tag destination address. |
| 0x18 | `LEN` | R/W | Length of the data to be encrypted or decrypted. Must be divisible by four. |
| 0x1C-0x28 | `NONCE[0-3]` | R/W | Nonce for the re-keying operation and the permutation-based encryption mode. |
| 0x2C | `SPONGE_CTRL` | R/W | Sponge control register to configure all permutation-based modes of operation. |
| 0x30-0x3C | `KEY0[0-3]` | R/W | Cipher key 0 for AES-128 ECB-, AES-128 leakage-resilient-, and permutation-based encryption mode. |
| 0x40-0x4C | `KEY1[0-3]` | R/W | Key 1 for the AES-128 XTS encryption mode and permutation-based authentication mode. |
| 0x50-0x60 | `SESS_KEY[0-3]` | R/W | Session key register used for re-keying only or to provide a session key if no re-keying mode is configured. |
| 0x64-0x70 | `CONST0[0-3]` | R/W | Constant $p_0$ for the AES-128 based leakage-resilient encryption mode. Used for the first round-key in AES-primitive operation. |
| 0x74-0x80 | `CONST1[0-3]` | R/W | Constant $p_1$ for the AES-128 based leakage-resilient encryption mode. Used for the second round-key in AES-primitive operation. |
| 0x84-0x94 | `SEED[0-4]` | R/W | 132-bit seed for the LFSR. |
| 0x98-0xB8 | `IV_REKEY[0-8]` | R/W | Initial value for permutation-based re-keying function *RK2*. |
| 0xBC-0xC8 | `IV_ENC[0-3]` | R/W | Initial value for permutation-based encryption mode. |
| 0xCC-0xEC | `IV_MAC[0-8]` | R/W | Initial value for permutation-based authentication mode. |
| 0xF0-0x120 | `USER_STATE[0-12]` | R/W | User state for primitive operations. `USER_STATE[0]` is also used for the padding configuration of permutation-based re-keying functions. |

### A.6.1 Register Description

We now describe special registers, which contain a bit configuration rather than being one single register such as the `LEN`, or `KEY_0` register.

**HWSTAT.** The status register *HWSTAT* of the accelerator is shown in Register A.2. This register allows the processing cores to monitor the operation and determines the current state of the accelerator. This register is read only.

Register A.2: HWSTAT. (0x000)




**CURR_CMD_ID** Returns the job ID for the currently operated job.

**NEXT_CMD_ID** Returns the job ID for the next job.

**E_AUTH_CFG** When logic one, a wrong authentication configuration for the current job request was performed.

**E_TAG_VERIF** When logic one, the tag verification for decryption has failed. Either the ciphertext, the nonce, or the tag has been corrupted.

**E_OUT_BNDS** When logic one, the memory source, or target range is out of the allowed bounds. This can happen for the TCDM range for encryption, or for the authentication tag, which is read and written to the TCDM.

**E_LEN_ZERO** When logic one, the length for the job started is zero.

**E_LEN_MOD** When logic one, the length for the job started is not divisible by four. The accelerator only supports encrypting data blocks which are multiple of 128-bit. Since the length is defined by 32-bit words, the length must be divisible by four.

**E_ENC_MODE** When logic one, an invalid encryption mode is configured.

**E_SHUF_MODE** When logic one, an invalid shuffling mode is configured.

**E_REKEY_MODE** When logic one, an invalid re-keying mode is configured.

**E_NO_MODE** Error bit indicating no mode is configured. This means neither a re-keying-, an encryption-, nor a primitive-mode is configured for the job request.

**QUEUE_FULL** When set to high, this bit indicates that the internal command queue is full. Any additional job request (`START`-bit is set to logic high) to the `HWCTRL` register will block the peripheral interface until there is again space in the command queue.

**FINISH** Logic high indicates a job has finished.

**BUSY** When logic one, this bit indicates the accelerator is busy with its current operation.

**HWCTRL.** The `HWCTRL` register controls and configures the *HWCrypt* accelerator. It allows the processing cores to configure the peripheral for a certain mode and to start an operation. When starting a new job, all other related registers must be configured in advance.

Register A.3: HWCTRL (0x004)

| RFU | PRIM_MODE | MASK_ORDER | RFU | SHUF_MODE | RFU | ENC_MODE | RFU | REKEY_MODE | FRESH_SEED | AUTH | ENC_DEC | START |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 30 26 | 26 24 | 23 16 | 15 14 | 13 12 | 11 | 10 8 | 7 | 6 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Reset

**PRIM_MODE** Primitive mode configuration. Valid values are listed below. Any other value will be treated as an error. For decryption operations, the `ENC_DEC` bit must be set separately. When using the Keccak-$f$[400] operation, the `ROUNDS_ENC` bit in the `SPONGE_CTRL` register must be configured separately.

**000** No primitive operation
**001** AES double-round encryption
**010** AES double-round decryption
**011** Last AES double-round encryption

**100** Last AES double-round decryption
**101** KECCAK-$f$[400] primitive operation

**MASK_ORDER** Masking order for polynomial re-keying. When set to `0x00` no masking is applied. Otherwise the masking order defined by this register is used. The maximum masking order is `255`.

**SHUF_MODE** Shuffling strategy for the partial multiplications of the polynomial re-keying unit. Valid values are listed below. Any other value will be treated as an error.

**00** No shuffling
**01** Shuffling of the start index
**10** Fisher-Yates shuffling

**ENC_MODE** Encryption mode. Valid values are listed below. Any other value will be treated as an error.

**000** No encryption
**001** AES-128 ECB encryption
**010** AES-128 XTS encryption
**011** AES-128 based leakage-resilient encryption
**100** Permutation-based encryption

**REKEY_MODE** Re-keying mode. Valid values are listed below. Any other value will be treated as an error.

**000** No re-keying
**001** Polynomial re-keying
**010** Polynomial re-keying with block cipher post-processing
**011** Key expansion for permutation-based re-keying *RK1*
**100** Key expansion for permutation-based re-keying *RK2*
**101** Permutation-based re-keying *RK1*
**110** Permutation-based re-keying *RK2*

**FRESH_SEED** Update the internal seed for the pseudo random number generator. The `SEED` register must be written before.

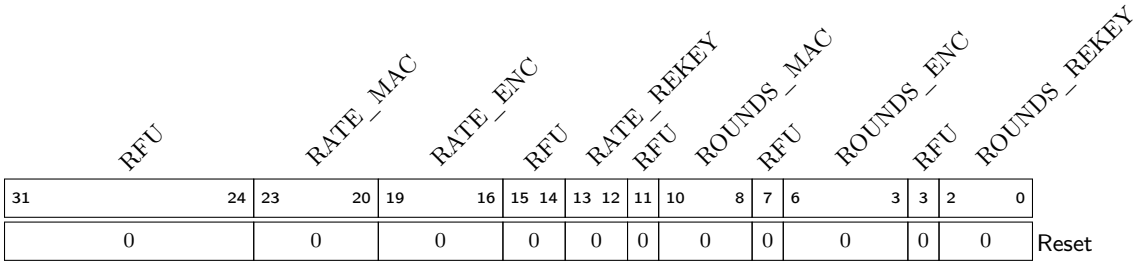
**AUTH** Enables the authentication mode. Only valid when using the permutation-based encryption mode and a dedicated re-keying mode. Otherwise an error is raised.

**ENC_DEC** When set to zero, encryption is performed. Else, decryption is performed.

**START** Starts a new operation with the current configuration.

**SPONGE_CTRL.** The `SPONGE_CTRL` register is used to provide a base-configuration of the internal KECCAK-$f$[400] permutation function, and for the sponge unit. The configuration of this register is required for all permutation-based operations.

Register A.4: SPONGE_CTRL. (0x038)




**RATE_MAC** Configures the absorbing rate for permutation-based authentication algorithm. Valid values are listed below. Any other value will be treated as an error.

**000** 1-bit
**001** 2-bit
**010** 4-bit
**011** 8-bit
**100** 16-bit
**101** 32-bit
**110** 64-bit
**111** 128-bit

**RATE_ENC** Configures the absorbing rate for permutation-based encryption. This configuration uses the same possible configuration as the bit-field `RATE_MAC`.

**RATE_REKEY** Rate for permutation-based re-keying function *RK2*. Valid values are listed below. Any other value will be treated as an error.

**0000** 1-bit
**0001** 2-bit
**0010** 4-bit
**0011** 8-bit
**1000** 144-bit

**ROUNDS_MAC** Defines the number of invoked round for the KECCAK-$f$[400] permutation function for the authentication algorithm. Valid values are listed below. Any other value will be treated as an error.

**000** 3 Rounds
**001** 6 Rounds
**010** 9 Rounds
**011** 12 Rounds
**100** 15 Rounds
**101** 18 Rounds
**110** 20 Rounds

**ROUNDS_ENC** Defines the number of invoked round for the KECCAK-$f$[400] permutation function for the encryption algorithm. It uses the same round values as defined for `ROUNDS_MAC`.

**ROUNDS_REKEY** Defines the number of invoked round for the KECCAK-$f$[400] permutation function for the re-keying functions *RK1* and *RK2*. It uses the same round values as defined for `ROUNDS_MAC`.

## A.7 Operation Modes

*HWCrypt* supports three different operating modes. First, encryption and decryption of the TCDM data are supported. Second, *HWCrypt* can perform standalone re-keying without encryption. Third, the accelerator is capable of doing a primitive operation of the AES round function and the permutation function. Each of these three operating modes supports different configurations which are described in the following sections.

All operations follow the same principle. First, the operating mode with all its required registers is configured. Second, the operation is started by writing the `HWCRTL` register with a proper configuration and the `START`-bit set to one. This starts the accelerator to perform the configured operation. When the job finishes, either the `FinishIrqEvent_SO` or `IntermediateIrqEvent_SO` interrupt is raised depending on the pending jobs in the command queue. Alternatively, the software can poll the `HWSTAT` register and read the `BUSY`-bit until the accelerator is finished and sets the `FINISH`-bit. The software can already start the next operation while the accelerator is still busy. A command queue stores the request and starts the new job when the current operation finishes. The queue can store four pending operations. If the queue is full, adding a new job to the queue by writing the `HWCTRL` with a set `START`-bit will block the peripheral interface.

### A.7.1 Re-keying Mode

The re-keying mode supports performing only a re-keying computation in hardware without encryption. The computed session key can be used in software for any other operation. To configure this mode, the `KEY_0` and the `NONCE` register must be written first.

Moreover, additional register configuration may be necessary depending on the selected re-keying mode. Table A.4 shows the `HWCTRL` register configuration for all possible re-keying modes, which are explained in detail in the following section. After finishing the operation, the computed session key can be read via the `SESS_KEY` register. Polynomial re-keying modes result in a 128-bit session key. Permutation-based session keys result in a 144-bit session key.

Table A.4: `HWCTRL` configuration for re-keying.

| Operation | `HWCTRL` value |
|---|---|
| No re-keying | 0x000001 |
| Polynomial re-keying | 0xmms011 |
| Polynomial re-keying with post-processing | 0xmms021 |
| Key expansion for permutation-based re-keying *RK1* | 0x000031 |
| Key expansion for permutation-based re-keying *RK2* | 0x000041 |
| Permutation-based re-keying *RK1* | 0x000051 |
| Permutation-based re-keying *RK2* | 0x000061 |

#### A.7.1.1 Polynomial Re-keying

For polynomial re-keying, the `HWCTRL` needs to be configured as defined in Table A.4 in the second row. This configuration contains two place-holders defined by $s$ and $m$. This place-holder $s$ defines the shuffling mode and can be set according to Register A.3. The second place-holder $m$ defines the masking order for the polynomial re-keying function and can be set to an 8-bit value to define the masking order.

**Relevant Register Configuration**

**HWCTRL** Configuration for the masking order and re-keying mode.

**KEY0** Master key.

**NONCE** Nonce for the re-keying operation.

#### A.7.1.2 Polynomial Re-keying with Block Cipher Post-Processing

The configuration for polynomial re-keying with block cipher post-processing as shown in Table A.4 is similar to the configuration of ordinary polynomial re-keying as defined in Section A.7.1.1. The only difference is a different re-keying mode as defined in the description of Register A.3.

**Relevant Register Configuration**

| | |
|---|---|
| **HWCTRL** | Configuration for the masking order and re-keying mode. |
| **KEY0** | Master key. |
| **NONCE** | Nonce for the re-keying operation. |

#### A.7.1.3 Key Expansion for Permutation-based Re-keying RK1

This operation performs the key expansion algorithm for the permutation-based re-keying mode *RK1*. To start this operation, the `SPONGE_CTRL` register needs to be configured before. In particular, the bit-field `ROUNDS_REKEY` need to be set to configure the number of required rounds of the permutation function.

**Relevant Register Configuration**

| | |
|---|---|
| **HWCTRL** | Configuration for the re-keying mode. |
| **KEY0** | Master key. |
| **KEY1** | Key for the permutation-based authentication mode. |
| **SPONGE_CTRL** | Round configuration for the `ROUNDS_REKEY` bit-field. |

#### A.7.1.4 Key Expansion for Permutation-based Re-keying RK2

This operation performs the key expansion algorithm for the permutation-based re-keying mode *RK2*. Moreover, this mode requires a more detailed configuration. First, the initial vector in register `IV_REKEY` needs to be written. Second, the bit-field `ROUNDS_REKEY` in the `SPONGE_CTRL` register needs to be configured before starting the expanding operation as defined in Table A.4.

**Relevant Register Configuration**

| | |
|---|---|
| **HWCTRL** | Configuration for the re-keying mode. |
| **KEY0** | Master key. |
| **KEY1** | Key for the permutation-based authentication mode. |
| **IV_REKEY** | Initial vector for permutation-based re-keying function *RK2*. |
| **SPONGE_CTRL** | Round configuration for the `ROUNDS_REKEY` bit-field. |

#### A.7.1.5 Permutation-based Re-keying RK1

This operation starts the permutation-based re-keying function *RK1*. The key expansion algorithm, as defined in Section A.7.1.3, must be executed before.

| **Relevant Register Configuration** | |
|---|---|
| **HWCTRL** | Configuration for the re-keying mode. |
| **NONCE** | Nonce for the re-keying operation. |
| **SPONGE_CTRL** | Round configuration for the `ROUNDS_REKEY` bit-field. |

#### A.7.1.6 Permutation-based Re-keying RK2

This operation starts the permutation-based re-keying function *RK2*. The key expansion algorithm as defined in Section A.7.1.4 must be executed before.

| **Relevant Register Configuration** | |
|---|---|
| **HWCTRL** | Configuration for the re-keying mode. |
| **NONCE** | Nonce for the re-keying operation. |
| **SPONGE_CTRL** | Round configuration for the `ROUNDS_REKEY` and `RATE_REKEY` bit-fields. |

### A.7.2 Encryption Mode

*HWCrypt* supports different encryption modes, which configurations are shown in Table A.5. For both the AES-based leakage-resilient encryption mode and the permutation-based encryption mode, the encryption operation can be combined with any re-keying mode as defined in Table A.4. Moreover, these two encryption modes support using a software-defined session key. In this case, the re-keying mode needs to be set to `0x0`, and the session key needs to be provided via the `SESS_KEY` register before starting a new operation. In this section, we describe the configuration of all encryption modes in detail. Furthermore, each configuration contains a section showing any additional relevant register configuration. All encryption modes support either encryption or decryption. For encryption, the `ENC_DEC`-bit is set to `0`, for decryption to `1`. Table A.5 defines all supported encryption modes.

Table A.5: HWCTRL configuration for encryption.

| Operation | HWCTRL value |
|---|---|
| No encryption | 0x000001 |
| AES-128 ECB encryption | 0x000101 |
| AES-128 XTS encryption | 0x000201 |
| AES-128-based leakage-resilient encryption | 0x000301 |
| Permutation-based encryption | 0x000401 |

#### A.7.2.1 AES-128 ECB Encryption

The accelerator supports AES-128 ECB encryption of the TCDM. Given the encryption key in register KEY0, TCDM data can be encrypted or decrypted.

**Relevant Register Configuration**

**HWCTRL** Encryption configuration.

**KEY0** Cipher key for encryption and decryption.

**TCDM_SRC** Source address of the data to be processed.

**TCDM_DST** Destination address of the data to be processed.

**LEN** Length of data to be processed. Must be divisible by four.

#### A.7.2.2 AES-128 XTS Encryption

*HWCrypt* supports AES-128 XTS encryption of data in the TCDM. If both keys KEY0 and KEY1 have the same value, XEX encryption is performed. The CONST0 register is used to act as the block number to compute the address-dependent tweak.

**Relevant Register Configuration**

**HWCTRL** Encryption configuration.

**KEY0** Cipher key for encryption and decryption.

**KEY1** Cipher key for tweak encryption.

**CONST0** Block number used for the tweak computation.

**TCDM_SRC** Source address of the data to be processed.

**TCDM_DST** Destination address of the data to be processed.

**LEN** Length of data to be processed. Must be divisible by four.

#### A.7.2.3 AES-128 based Leakage-Resilient Encryption

The accelerator supports a leakage-resilient encryption mode, in which every block is encrypted differently. This encryption mode can be configured with any re-keying mode as defined in Section A.7.1. The registers `CONST0` and `CONST1` serve as the constants $p_0$ and $p_1$ for the leakage-resilient encryption algorithm. Their recommended values are `0` for `CONST0` and `1` for `CONST1` respectively.

**Relevant Register Configuration**

| | |
|---|---|
| **HWCTRL** | Encryption configuration. |
| **CONST0** | Constant $p_0$ for the leakage-resilient mode of operation. |
| **CONST1** | Constant $p_1$ for the leakage-resilient mode of operation. |
| **TCDM_SRC** | Source address of the data to be processed. |
| **TCDM_DST** | Destination address of the data to be processed. |
| **LEN** | Length of data to be processed. Must be divisible by four. |

#### A.7.2.4 Permutation-based Encryption

The permutation-based encryption mode supports a leakage-resilient encryption of the TCDM with DPA-safety by design. Furthermore, it supports a configurable data processing rate to trade-off the security versus the performance. Register A.4 defines the processing rate for encryption. The lower the rate is, the higher is the capacity of the sponge construction. This means that with the reduced rate, which equals a reduced throughput, the security is increased. Moreover, the encryption mode supports a different number of invoked rounds of the permutation function as defined in Register A.4.

**Authentication** Permutation-based encryption supports authentication of the ciphertext. To use this mode the `AUTH`-bit in the `HWCRTL` register needs to be set. Authentication requires the second key to be written in advance to the `KEY1` register. When using a permutation-based re-keying mode, the key also needs to be expanded as defined in Section A.7.1.3 and Section A.7.1.4. For encryption, the authentication function computes an authentication tag, which is written to the TCDM address defined by the `TCDM_TDST` register. Decryption with authentication requires the `TCDM_TSRC` register to point to the TCDM memory where the 128-bit authentication tag for the ciphertext is located. Furthermore, authentication requires the `ROUNDS_MAC` and `RATE_MAC` bit-fields in the `SPONGE_CTRL` register to be configured properly. Authentication supports all re-keying functions except the software-defined session keys, in which no re-keying mode is configured.

**Relevant Register Configuration**

| | |
|---|---|
| **HWCTRL** | Encryption and authentication configuration. |
| **SPONGE_CTRL** | Sponge configuration for round- and rate-configuration for encryption and authentication. |
| **IV_ENC** | Initial vector used for the permutation-based encryption mode. |
| **IV_MAC** | Initial vector used for the permutation-based authentication mode. |
| **TCDM_SRC** | Source address of the data to be processed. |
| **TCDM_DST** | Destination address of the data to be processed. |
| **LEN** | Length of data to be processed. Must be divisible by four. |
| **TCDM_TSRC** | Source address of the authentication tag used for decryption. |
| **TCDM_TDST** | Target address for authentication tag computed during encryption. |
| **NONCE** | Nonce used in the encryption mode. |
| **USER_STATE0** | Bits `[15:0]` define the padding of the nonce used in this encryption mode. |

### A.7.3 Primitive Mode

The primitive mode aims to make internally used primitive instances accessible via the peripheral interface. This mode of operation is not optimized for performance rather it supports a hardware accelerated software implementation of an arbitrary algorithm which uses any of the provided primitives.

Table A.6: `HWCTRL` configuration for primitive operation.

| Operation | `HWCTRL` value |
|---|---|
| Double AES round encryption | 0x1000001 |
| Double AES round decryption | 0x2000003 |
| Last double AES round encryption | 0x3000001 |
| Last double AES round decryption | 0x4000003 |
| Keccak-$f$[400] | 0x5000001 |

#### A.7.3.1 AES Round Operation

Although the AES-128 based encryption mode already supports ECB encryption, we further extend the flexibility of the accelerator by making the AES round function available via the peripheral interface. This allows the user to implement any AES round-based algorithm with hardware acceleration. The primitive operation of AES distinguishes between four different operating modes, which are listed in Table A.6. A normal AES double round and the last AES double round (without the *MixColumns* operation) are available for both encryption and decryption via this primitive interface. To program a primitive operation, the software needs first to write the `USER_STATE` registers. `USER_STATE[0-3]` define the first 128-bit word, whereas `USER_STATE[4-7]` define the second 128-bit word. Furthermore, the software needs to write the registers `CONST0` and `CONST1`, which define the round-keys for the first and the second round, respectively. Then, software must trigger the operation via writing the proper configuration to the `HWCTRL` register. Table A.6 indicates the configuration for the supported modes. These configurations already contain a set start bit. After finishing the operation, the two words are readable via the `USER_STATE` register.

| **Relevant Register Configuration** | |
|---|---|
| **HWCTRL** | Configuration of the primitive operation. |
| **USER_STATE[0-3]** | First 128-bit AES word. |
| **USER_STATE[4-7]** | Second 128-bit AES word. |
| **CONST0** | Round-key for the first round function. |
| **CONST1** | Round-key for the second round function. |

#### A.7.3.2 Keccak-$f$[400] Permutation

The sponge unit uses the Keccak-$f$[400] permutation function internally. This permutation function is made accessible via the peripheral interface, which makes it possible to implement a hardware accelerated sponge construction in software, such as the Keccak hash algorithm. For this computation, the 400-bit state is written to the registers `USER_STATE[0-12]`. Furthermore, this mode uses the `ROUNDS_ENC` configuration from the `SPONGE_CTRL` register to determine the number of rounds. Register A.4 defines the possible configuration values for all available round configurations. The operation is started by writing the `HWCRTL` register with the value for Keccak-$f$[400] primitive operation as defined by Table A.6. After finishing the operation, the computed state is readable in register `USER_STATE[0-12]`.

**Relevant Register Configuration**

| | |
|---|---|
| **HWCTRL** | Configuration of the primitive operation. |
| **SPONGE_CTRL** | Sponge configuration number of rounds. |
| **USER_STATE[0-12]** | 400-bit state for the permutation function. |

# Bibliography


[1] C. Dobraunig, M. Eichlseder, S. Mangard, F. Mendel, and T. Unterluggauer, "PASEC: Keccak," 2016.

[2] M. Carlie and C. Valasek, "Remote Car Hacking," http://illmatics.com/Remote%20Car%20Hacking.pdf, 2015.

[3] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest We Remember: Cold Boot Attacks on Encryption Keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[4] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer, *Topics in Cryptology - CT-RSA 2016: The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings.* Cham: Springer International Publishing, 2016, ch. ECDH Key-Extraction via Low-Bandwidth Electromagnetic Attacks on PCs, pp. 219–235. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-29485-8_13

[5] D. Lampret, C.-M. Chen, M. Mlinar, J. Rydberg, M. Ziv-Av, C. Ziomkowski, G. McGary, B. Gardner, R. Mathur, and M. Bolado, "OpenRISC 1000 Architecture Manual," *Rev*, vol. 1, p. 15, 2007.

[6] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54, May 2014. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[7] NIST, *Advanced Encryption Standard (AES) (FIPS PUB 197)*, National Institute of Standards and Technology, Nov. 2001.

[8] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Permutation-based encryption, authentication and authenticated encryption," *Directions in Authenticated Ciphers*, 2012.

[9] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak sponge function family main document," *Submission to NIST (Round 2)*, vol. 3, p. 30, 2009.

[10] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology — CRYPTO' 99*, ser. Lecture Notes in Computer Science, M. Wiener, Ed. Springer Berlin Heidelberg, 1999, vol. 1666, pp. 388–397. [Online]. Available: http://dx.doi.org/10.1007/3-540-48405-1_25

[11] P. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Advances in Cryptology — CRYPTO '96*, ser. Lecture Notes in Computer Science, N. Koblitz, Ed. Springer Berlin Heidelberg, 1996, vol. 1109, pp. 104–113. [Online]. Available: http://dx.doi.org/10.1007/3-540-68697-5_9

[12] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The EM side—channel(s)," in *Cryptographic Hardware and Embedded Systems-CHES 2002*. Springer, 2003, pp. 29–45.

[13] K. Gandolfi, C. Mourtel, and F. Olivier, *Cryptographic Hardware and Embedded Systems — CHES 2001: Third International Workshop Paris, France, May 14–16, 2001 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, ch. Electromagnetic Analysis: Concrete Results, pp. 251–261. [Online]. Available: http://dx.doi.org/10.1007/3-540-44709-1_21

[14] Genkin, Daniel and Shamir, Adi and Tromer, Eran, *RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 444–461. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-44371-2_25

[15] D. Boneh, R. A. DeMillo, and R. J. Lipton, *Advances in Cryptology — EUROCRYPT '97: International Conference on the Theory and Application of Cryptographic Techniques Konstanz, Germany, May 11–15, 1997 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, ch. On the Importance of Checking Cryptographic Protocols for Faults, pp. 37–51. [Online]. Available: http://dx.doi.org/10.1007/3-540-69053-0_4

[16] D. Boneh, A. R. DeMillo, and J. R. Lipton, "On the Importance of Eliminating Errors in Cryptographic Computations," *Journal of Cryptology*, vol. 14, no. 2, pp. 101–119, 2000. [Online]. Available: http://dx.doi.org/10.1007/s001450010016

[17] S. Mangard, "A Simple Power-Analysis (SPA) Attack on Implementations of the AES Key Expansion," in *Information Security and Cryptology—ICISC 2002*. Springer, 2002, pp. 343–358.

[18] T. Messerges, "Using Second-Order Power Analysis to Attack DPA Resistant Software," in *Cryptographic Hardware and Embedded Systems—CHES 2000*. Springer, 2000, pp. 27–78.

[19] J. Waddle and D. Wagner, *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, ch. Towards Efficient Second-Order Power Analysis, pp. 1–15. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-28632-5_1

[20] E. Brier, C. Clavier, and F. Olivier, *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, ch. Correlation Power Analysis with a Leakage Model, pp. 16–29. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-28632-5_2

[21] J.-M. Cioranesco, J.-L. Danger, T. Graba, S. Guilley, Y. Mathieu, D. Naccache, and X. T. Ngo, "Cryptographically Secure Shields," in *Hardware-Oriented Security and Trust (HOST), 2014*. IEEE, 2014, pp. 25–31.

[22] K. Tiri and I. Verbauwhede, "Securing Encryption Algorithms against DPA at the Logic Level: Next Generation Smart Card Technology," in *Cryptographic Hardware and Embedded Systems-CHES 2003*. Springer, 2003, pp. 125–136.

[23] T. Popp and S. Mangard, "Masked Dual-Rail Pre-charge Logic: DPA-Resistance Without Routing Constraints," in *Cryptographic Hardware and Embedded Systems – CHES 2005*, ser. Lecture Notes in Computer Science, J. Rao and B. Sunar, Eds. Springer Berlin Heidelberg, 2005, vol. 3659, pp. 172–186. [Online]. Available: http://dx.doi.org/10.1007/11545262_13

[24] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer Science & Business Media, 2008, vol. 31.

[25] M. Medwed, F.-X. Standaert, J. Großschädl, and F. Regazzoni, "Fresh Re-keying: Security against Side-Channel and Fault Attacks for Low-Cost Devices," in *Progress in Cryptology – AFRICACRYPT 2010*, ser. Lecture Notes in Computer Science, D. Bernstein and T. Lange, Eds. Springer Berlin Heidelberg, 2010, vol. 6055, pp. 279–296. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12678-9_17

[26] K. Itoh, M. Takenaka, and N. Torii, "DPA Countermeasure Based on the "Masking Method"," in *Information Security and Cryptology—ICISC 2001*. Springer, 2002, pp. 440–456.

[27] M.-L. Akkar and C. Giraud, "An Implementation of DES and AES, Secure against Some Attacks," in *Cryptographic Hardware and Embedded Systems—CHES 2001*. Springer, 2001, pp. 309–318.

[28] K. Schramm and C. Paar, "Higher Order Masking of the AES," in *Topics in Cryptology–CT-RSA 2006*. Springer, 2006, pp. 208–225.

[29] N. Pramstaller, F. K. Gurkaynak, S. Haene, H. Kaeslin, N. Felber, and W. Fichtner, "Towards an AES Crypto-Chip Resistant to Differential Power Analysis," in *Solid-State Circuits Conference, 2004. ESSCIRC 2004. Proceeding of the 30th European*. IEEE, 2004, pp. 307–310.

[30] S. Mangard, N. Pramstaller, and E. Oswald, "Successfully Attacking Masked AES Hardware Implementations," in *Cryptographic Hardware and Embedded Systems–CHES 2005*. Springer, 2005, pp. 157–171.

[31] F. Regazzoni, Y. Wang, F.-X. Standaert *et al.*, "FPGA Implementations of the AES Masked Against Power Analysis Attacks," in *Proceedings of COSADE 2011, International Workshop on Side-Channel Analysis and Secure Design*, 2011.

[32] D. Hwang, K. Tiri, A. Hodjat, B.-C. Lai, S. Yang, P. Schaumont, and I. Verbauwhede, "AES-Based Cryptographic and Biometric Security Coprocessor IC in 0.18-um CMOS Resistant to Side-Channel Power Analysis Attacks ," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 4, pp. 781–792, 2006.

[33] S. Belaïd, F. De Santis, J. Heyszl, S. Mangard, M. Medwed, J.-M. Schmidt, F.-X. Standaert, and S. Tillich, "Towards Fresh Re-Keying with Leakage-Resilient PRFs: Cipher Design Principles and Analysis," *Journal of Cryptographic Engineering*, vol. 4, no. 3, pp. 157–171, 2014. [Online]. Available: http://dx.doi.org/10.1007/s13389-014-0079-5

[34] P. C. Kocher, "Leak-Resistant Cryptographic Indexed Key Update," Mar. 25 2003, US Patent 6,539,092.

[35] Atmel, "Datasheet SAMA5D4 Series," http://www.atmel.com/images/atmel-11238-32-bit-cortex-a5-microcontroller-sama5d4_datasheet.pdf, 2015.

[36] Intel, "Intel Advanced Encryption Standard (AES) New Instructions Set," https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf, 2012.

[37] A. Bogdanov, M. M. Lauridsen, and E. Tischhauser, "Comb to Pipeline: Fast Software Encryption Revisited," in *Fast Software Encryption*. Springer, 2015, pp. 150–171.

[38] Rambus Cryptography Research, "Licensed Countermeasures," https://www.rambus.com/security/dpa-countermeasures/licensed-countermeasures/, 2016.

[39] NXP Semiconductors, Business Line Identification, "P5CT072/ P5CC072/ P5CN072/ P5CD072/P5CD036/P5CN036 V0S Secure SmartCard Controller," https://www.rambus.com/security/dpa-countermeasures/licensed-countermeasures/, 2008.

[40] R. A. Fisher, F. Yates *et al.*, "Statistical Tables for Biological, Agricultural and Medical Research." *Statistical Tables for Biological, Agricultural and Medical Research.*, no. Ed. 3., 1949.

[41] D. E. Knuth, "Seminumerical Algorithms, The Art of Computer Programming, Vol. 2," 1981.

[42] C. Dobraunig, M. Eichlseder, S. Mangard, and F. Mendel, "On the Security of Fresh Re-keying to Counteract Side-Channel and Fault Attacks," in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, M. Joye and A. Moradi, Eds. Springer International Publishing, 2015, vol. 8968, pp. 233–244. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-16763-3_14

[43] F.-X. Standaert, O. Pereira, and Y. Yu, "Leakage-Resilient Symmetric Cryptography under Empirically Verifiable Assumptions," in *Advances in Cryptology - CRYPTO 2013*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 8042, pp. 335–352. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40041-4_19

[44] C. Dobraunig, M. Eichlseder, S. Mangard, F. Mendel, and T. Unterluggauer, "PASEC: Securing Authenticated Encryption Against Passive Side-Channel Attacks By Design," 2016.

[45] O. Goldreich, S. Goldwasser, and S. Micali, "How to Construct Random Functions," *Journal of the ACM (JACM)*, vol. 33, no. 4, pp. 792–807, 1986.

[46] M. Dworkin, "Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices," *NIST Special Publication*, vol. 800, 2010.

[47] Kingston Technology, "Encrypted Drives," http://www.kingston.com/en/usb/encrypted_security, 2016.

[48] Samsung Semiconductor, Inc., "Samsung Solid State Drives," http://www.samsung.com/us/business/oem-solutions/pdfs/selfencryptingssd-042011.pdf, 2016.

[49] P. Rogaway, "Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC," in *Advances in Cryptology - ASIACRYPT 2004*, ser. Lecture Notes in Computer Science, P. Lee, Ed. Springer Berlin Heidelberg, 2004, vol. 3329, pp. 16–31. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30539-2_2

[50] R. Schilling, "Memcrypt - A Fully Transparent Memory Encryption," 2015, Master Project.

[51] I. Xilinx, "Efficient Shift Registers, LFSR Counters, and Long Pseudo- Random Sequence Generators," http://www.xilinx.com/support/documentation/application_notes/xapp052.pdf, 1996.

[52] J. Wolkerstorfer, "An ASIC Implementation of the AES MixColumn operation," in *Austrochip 2001*, 2001, pp. 129–132.

[53] A. Ronacher, "Jinja2 - A Full Featured Template Engine in Python," http://jinja.pocoo.org/, 2014.

[54] D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright, "Fast Software AES Encryption," in *Fast Software Encryption*. Springer, 2010, pp. 75–93.

[55] P. Neal, "The Generalised Coupon Collector Problem," *Journal of Applied Probability*, pp. 621–629, 2008.

[56] E. Biham and A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," in *Advances in Cryptology — CRYPTO '97*, ser. Lecture Notes in Computer Science, J. Kaliski, BurtonS., Ed. Springer Berlin Heidelberg, 1997, vol. 1294, pp. 513–525. [Online]. Available: http://dx.doi.org/10.1007/BFb0052259

[57] C. E. Dobraunig, F. Koeune, S. Mangard, F. Mendel, and F.-X. Standaert, "Towards Fresh and Hybrid Re-Keying Schemes with Beyond Birthday Security," in *CARDIS*, ser. LNCS, N. Homma and M. Medwed, Eds. Springer, 2015, in press.