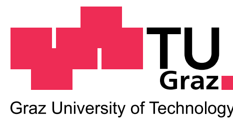


Christian Pendl
christian.pendl@student.tugraz.at

100 Gbit/s Authenticated Encryption Based on Quantum Key Distribution

Master's Thesis

to achieve the university degree of
Master of Science

Master's degree programme: Telematics

submitted to

Graz University of Technology

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

Integrated Systems Laboratory (IIS)
Swiss Federal Institute of Technology
Gloriastrasse 35
CH-8092 Zürich, Switzerland

Supervisors: Thomas Korak, TU Graz
Christoph Keller, ETH Zürich
Michael Mühlberghuber, ETH Zürich

Assessors: Prof. Stefan Mangard, TU Graz
Prof. Hubert Kaeslin, ETH Zürich

December, 2014

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

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

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

Englische Fassung:

STATUTORY DECLARATION

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

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

Acknowledgements

I would like to thank the members of the Institute for Applied Information Processing and Communications (IAIK) at the Graz University of Technology (TUG) and the Swiss Federal Institute of Technology (ETH) Zürich for giving me the chance to do my thesis abroad. I would also like to thank my supervisors Michael Hutter and Thomas Korak from the TUG, and Michael Mühlberghuber and Christoph Keller from the ETH Zürich and all members of the Microelectronics Design Center from the Department of Information Technology and Electrical Engineering (D-ITET) at ETH for their remarkable support during the months of working on this thesis. Particular thanks go to Michael Hutter for his excellent advice on writing and presenting scientific papers.
Last but not least, the biggest thanks go to my family. They showed almost endless patience and supported me during every phase of my studies, especially during the time of writing this master thesis. I know that my decisions and my acting were not always easy to understand and accept, but they were always behind me. Because of this I would like to dedicate this work to my family.

Abstract

The QCrypt project aims at providing a future proof cryptographic engine and to considerably improve cryptography on both, the key distribution level and the encryption level. It combines Quantum Key Distribution (QKD) with classical cryptographic primitives to provide authenticated encryption (AE). QKD is a secure way to generate and distribute cryptographic keys based on the fundamental laws of quantum mechanics. In today's standard applications data exchange rates continue to increase. The Ethernet standard IEEE 802.3ba already allows data rates of up to 100 Gbit/s. Consequently, a future proof encryption engine has to support up to 100 Gbit/s.

This work is part of the QCrypt project with the goal to extend the currently applied AE engine. The existing engine is based on the Galois Counter Mode of Operation (GCM) with AES as the underlying block cipher. Alternatives for both, the block cipher as well as the mode of operation should be evaluated. Besides exploring more efficient hardware implementations, this work is also motivated by providing an alternative AE scheme, in case successful attacks against primitives used in the existing system are developed. The main design goal is to achieve high throughput on FPGA platforms and to be compatible and easily interchangeable with the existing AE engine. The Serpent block cipher and the Offset CodeBook mode of operation figured out to be the best alternatives to the current system. Therefore, we evaluate the Serpent block cipher and the OCB mode of operation and provide results of hardware implementations for different mode of operation/block cipher combinations, namely: GCM-Serpent, OCB-AES and OCB-Serpent. All three authenticated-encryption engine variants are capable of providing authenticated encryption at a rate of over 100 Gbit/s.

To the best of our knowledge, GCM-Serpent and OCB-Serpent are the first hardware architectures targeting high-throughput authenticated encryption that are based on a block-cipher other than AES. Additionally, no design using AES as a block cipher, that is capable of reaching throughputs of over 100 Gbit/s, and that is based on a different mode of operation than GCM has been published so far. Our fastest design is based on OCB-Serpent and reaches a throughput of 136 Gbit/s at a maximum frequency of 267 MHz when using four cipher cores in parallel. This design outperforms all GCM-AES implementations available on FPGAs to date. Furthermore, our results show, that OCB is twice as efficient compared to GCM when taking the throughput/area ratio as an indicator.

Keywords: Cryptography, Authenticated encryption, GCM, OCB, Serpent, AES, FPGA, Quantum key distribution

Kurzfassung

Das QCrypt Projekt hat sich das Ziel gesetzt, eine zukunftssichere Kryptografielösung anzubieten und die Kryptografie sowohl hinsichtlich der Verschlüsselung als auch hinsichtlich der Schlüsselverteilung zu verbessern. Das System kombiniert die Quantenschlüsselverteilung, welche auf Basis der Gesetzte der Quantenmechanik eine sichere Methode zur Verteilung kryptografischer Schlüssel bietet, mit klassischen kryptografischen Methoden um gleichzeitig Authentifizierung und Verschlüsselung von Daten anzubieten. Die heute in Standardanwendungen verwendeten Datenraten steigen beständig und der Ethernet Standard IEEE 802.3ba erlaubt bereits Datenraten bis zu 100 Gbit/s. Als Folge muss eine zukunftssichere Lösung zur Authentifizierung und Verschlüsselung Datenraten von 100 Gbit/s unterstützen.

Diese Arbeit ist Teil des QCrypt Projekts und untersucht alternative Blockchiffren und Betriebsmodi zur Authentifizierung und Verschlüsselung zum existierenden AE-System. Dieses basiert auf dem Galois Counter Mode of Operation (GCM) mit AES als zu Grunde liegender Blockchiffre. Neben der Erforschung von effizienteren Hardwareimplementierungen, wird die Arbeit auch von dem Ziel angetrieben, alternative Methoden zur Authentifizierung und Verschlüsselung anzubieten, falls erfolgreiche Attacken gegen die derzeit verwendeten Methoden gefunden werden sollten. Das Hauptdesignziel ist es, hohen Datendurchsatz auf FPGA-Plattformen zu erreichen und gleichzeitig kompatibel und einfach austauschbar zur existierenden AE-Implementierung zu sein. Im Rahmen dieser Arbeit evaluieren wir die Blockchiffre Serpent und den Betriebsmodus Offset CodeBook (OCB) und präsentieren die Ergebnisse von Implementierungen verschiedener Kombinationen von Betriebsmodi und Blockchiffren. Diese sind: GCM-Serpent, OCB-AES und OCB-Serpent. Alle drei Varianten erreichen Datenraten von über 100 Gbit/s.

Laut aktueller Recherche sind GCM-Serpent und OCB-Serpent die ersten Hardwarearchitekturen, welche auf hohen Datendurchsatz abzielen, die nicht auf AES basieren. Zudem ist bisher kein auf AES-basiertes Design veröffentlicht worden, das einen Durchsatz von 100 Gbit/s erreicht und auf einem anderen Betriebsmodus als GCM basiert. Unser schnellstes Design basiert auf OCB-Serpent und erreicht einen Datendurchsatz von 136 Gbit/s bei einer maximalen Frequenz von 267 MHz. Es übertrifft somit alle bisher auf FPGA verfügbaren GCM-AES Implementierungen. Außerdem zeigen unsere Ergebnisse, dass der OCB-Modus doppelt so effizient wie der GCM-Modus ist.

Stichwörter: Kryptografie, Authentifizierung, Verschlüsselung, GCM, OCB, Serpent, AES, FPGA, Quantenschlüsselaustausch

Contents

1 Introduction **1**
1.1 Related Work . . . 2
1.2 Outline . . . 4

2 Cryptographic Goals **5**
2.1 The Major Cryptographic Goals . . . 5
2.2 Communication Model . . . 6
2.3 Attack Model . . . 7

3 Cryptographic Primitives **9**
3.1 Encryption Schemes . . . 9
3.1.1 Public-Key Encryption Schemes . . . 10
3.1.2 Symmetric Encryption Schemes . . . 10
3.1.3 Symmetric versus Asymmetric Encryption . . . 12
3.2 Authentication Schemes . . . 13
3.2.1 Unkeyed Hash Functions . . . 13
3.2.2 Keyed Hash Functions . . . 14
3.3 Quantum Cryptography . . . 15
3.3.1 Example of the BB84 Protocol . . . 16
3.3.2 Other Protocols . . . 17
3.3.3 Security of Quantum Cryptography . . . 18
3.3.4 Summary and Discussion . . . 19
3.4 Conclusion . . . 19

4 Key-Distribution Problem **20**

5 Block Ciphers Used for the Thesis **23**
5.1 Rijndael (AES) . . . 23
5.1.1 Data Representation . . . 24
5.1.2 Round Transformation . . . 24
5.1.3 Round-Key Generation . . . 27
5.2 Serpent . . . 27
5.2.1 Data Representation . . . 28
5.2.2 The Cipher . . . 28
5.2.3 Key Scheduler . . . 30

6 Authenticated Encryption **32**
6.1 Approaches to Achieve Authenticated Encryption 32
6.2 Properties and Features of Authenticated Encryption Algorithms 33
6.3 CCM 35
6.4 CWC 36
6.5 Galois Counter Mode of Operation 36
6.5.1 Authenticated Encryption 36
6.5.2 Authenticated Decryption 37
6.5.3 GHASH 38
6.6 Offset CodeBook (OCB) 39
6.6.1 Authenticated Encryption 39
6.6.2 Authenticated Decryption 42

7 Reconfigurable Hardware **44**
7.1 Introduction to Reconfigurable Hardware 44
7.2 Field-Programmable Gate Arrays (FPGAs) 45
7.2.1 Logic Array Blocks (LABs) and Arithmetic Logic Modules (ALMs) . 45
7.2.2 Configurable I/O Blocks 46
7.2.3 Programmable Interconnect and Clock Circuitry 46
7.2.4 Programming and Configuration of FPGAs 47

8 Overview of the Hardware Platform **48**
8.1 System Overview 48
8.2 The Quantum Key-Distribution System 48
8.3 The Fast-Encryptor System 49
8.4 The Altera Stratix IV EP4S100G5 FPGA 50
8.4.1 Logic Array Blocks (LABs) and Adaptive Logic Modules (ALMs) . . 50

9 The Existing Authenticated-Encryption Engine **52**
9.1 Multi-Core AES Design 53
9.2 The Parallel Pipelined GHASH Design 53
9.3 GCM Design 56

10 Analysis of Alternatives to the Existing System **58**
10.1 Analysis of Requirements 58
10.2 Alternative Block Cipher 61
10.2.1 AES 61
10.2.2 Serpent 62
10.2.3 Comparison and Choice of an Alternative Blockcipher 63
10.3 Alternatives Modes for Authenticated Encryption 64
10.3.1 GCM 66
10.3.2 OCB 66
10.3.3 Comparison and Choice of an Alternative Mode for Authenticated Encryption 67

11 Serpent - Implementation of an Alternative Block Cipher **69**
11.1 Single-Core Design 69
11.1.1 The Substitution Stage 69
11.1.2 The Linear Transformation 70
11.1.3 The Key Scheduler 71
11.2 Multi-Core Design 73
11.3 Summary 74

12 OCB - An alternative for Authenticated Encryption **75**
12.1 Encyption 75
12.2 Decryption 78
12.3 OCB-AES 79
12.4 OCB-Serpent 80

13 Simulation and Verification **82**
13.1 File-based Simulation 82
13.2 FPGA Testbench Design 84

14 Results **86**
14.1 Block Ciphers 86
14.1.1 Single Core 87
14.1.2 Multi Core 88
14.2 Modes for Authenticated Encryption 90
14.3 Comparison with Related Work 93

15 Conclusion **96**
15.1 Future Work 97
15.2 Outlook 98

Appendices **99**

A Definitions **99**
A.1 Abbreviations 99
A.2 Used Symbols 100

B Supplementary Material **101**
B.1 Serpent 101
B.2 OCB 104

C Original Assignment **105**

Bibliography **111**

Chapter 1

Introduction

Today, the dominating part of communication is transmitted over public channels. The amount of data transferred as well as the communication speed increases continuously. The Ethernet standard IEEE 802.3ba already allows data rates of up to 100 Gbit/s. However, all this communication needs to be protected by cryptographic primitives to ensure the privacy and authenticity of the data transmitted.

The contribution of this work is part of the QCrypt project [85] which aims at providing a future-proof, secure, high-speed communication platform based on a quantum key-distribution (QKD) system. The system can be divided into two parts: the quantum key-distribution system and the fast-encryptor system. The fast-encryptor system is based on an Altera Stratix IV EP4S100G5 FPGA. It handles the high-speed network communication and provides authenticated encryption (AE) for the data stream at a throughput of 100 Gbit/s using the user-keys supplied by the QDK-system. This stream is then transmitted to the communication partner over a public 100 Gbit/s channel. The existing authenticated-encryption engine is based on the Galois Counter Mode of Operation (GCM) with four parallel AES cores as underlying block cipher and is referred to as GCM-AES.

The main contribution of this work was to elaborate possible alternatives to the existing authenticated-encryption engine, that are capable of encrypting at data rates of over 100 Gbit/s. On the one hand the idea was to be prepared for unexpected security flaws in the existing system by providing alternative subcomponents and on the other hand there was a desire to possibly find a better solution compared to the existing system. In order to be able to identify promising alternatives we analyzed existing the authenticated-encryption engine and identified the requirements the authenticated-encryption engine has to fulfill. Then, we evaluated alternative subcomponents and algorithms in regard to the previously defined requirements. Finally, the most promising candidates were designed, implemented in hardware and tested on the target system. We laid a strong emphasis to ensure easy integration into the existing system, high throughput, an efficient use of resources, low latency and minimal delays. During the evaluation of alternatives to the AES block cipher, Serpent emerged as a promising candidate and was then implemented in hardware. In order to reach the desired throughput of 100 Gbit/s, 33 pipeline stages had to be introduced and four cipher cores had to be used in parallel. The resulting architecture is able to achieve a throughput of 140 Gbit/s at a maximum frequency of 275 MHz which is an increase of 8.3 % compared to the existing AES architecture. The

design requires 33,067 arithmetic logic modules (ALMs) and does not use any block RAM memory (BRAM). Based on the multi-core Serpent design, we developed a mode for AE that uses GCM with Serpent as the underlying block cipher. This design reached a throughput of 104 Gbit/s at a maximum frequency of 203 MHz while consuming 56,474 ALMs of a total of 212,480 ALMs available on the target platform. In addition we have chosen the Offset CodeBook (OCB) mode in its third version as an alternative mode for authenticated encryption. The OCB-encryption architecture is based on four parallel block cipher cores, while the OCB-decryption architecture uses four cipher cores in parallel for decryption and an additional cipher core that is in encryption mode. When using AES as the underlying block cipher OCB achieves a throughput of 112 Gbit/s at a maximum frequency of 220 MHz and only needs 10,060 ALMs for encryption and 11,614 ALMs for decryption. Compared to GCM-AES this is an increase of over 7 % in throughput. The fastest authenticated-encryption design is OCB-Serpent, which reaches a throughput of 136 Gbit/s at 276 MHz while consuming 29,506 ALMs for encryption and 33,891 ALMs for decryption. This is an increase in throughput of 27 % compared to GCM-AES. Overall, the throughput/area ratio of the OCB implementation is more than twice as high as of the existing GCM architecture.

Parts of this thesis have been published in [74] and [111] and in [73] Muehlberghuber et al. present a follow-up work based on this work. For a full description of the original assignment of this work please refer to Appendix C.

1.1 Related Work

In the early 1990s it became obvious that the security of the *Data Encryption Standard (DES)* [43] was not future proof any more. Therefore, in 1997, the National Institute of Standards and Technology (NIST) together with the industry and the cryptographic community started the Advanced Encryption Standard (AES) competition in search for a new standardized symmetric-key scheme. In 1999, after multiple rounds of public discussion, five candidates were presented that made it to the final round: *MARS* [26], *Rivest Cipher 6 (RC6)* [90], Rijndael [31], *Serpent* [10] and *Twofish* [98]. Those five candidates, also called AES finalists, were then extensively analyzed and explored in regard to their security, performance in software and performance in hardware. Since Rijndael was announced as the winner of the AES competition, it is referred to as AES, and most of the high speed implementations of block ciphers target this algorithm.

In 2011, Ali et al. [6] published an AES design accomplishing a throughput of 36.2 Gbit/s on a Altera Stratix-2 FPGA. In order to improve the throughput they used full loop unrolling with two pipeline stages per AES round and employed offline key calculation. Furthermore, they combined the SubBytes, the ShiftRows and the MixColumns operations, which are the fundamental building blocs of the AES round function, into a single step and precomputed a look-up table (LUT). This reduces the number of stages in an AES round and minimizes the critical path. Using the same principle, Cai, Sun, and Liu [28] reached a throughput of 40.96 Gbit/s on a Xilinx Spartan-6. They only introduced one pipeline stage per AES round and could thus reduce the latency to 10 clock cycles. Qiong and Jianwu [86] present an AES implementation reaching a throughput of 62.8 Gbit/s on an Altera Stratix-3 FPGA. Their design is fully unrolled with three pipeline stages for each round of AES and makes extensive use of block RAM (BRAM). In 2013, Liu, Xu, and Yuan [65] described an fully unrolled AES design with two pipeline stages for each round of AES that reaches a throughput of 66.1 Gbit/s on a Xilinx Virtex-7 FPGA without using BRAM. A throughput of 70 Gbit/s is reached on a Virtex-5 by Soliman and Abozaid [102] with a fully unrolled design that

uses two pipeline stages per AES round. Farashahi, Rashidi, and Sayedi [41] reached a throughput of 86 Gbit/s on a Xilinx Virtex-5 by introducing numerous pipeline stages.

Apart from AES, most of the research on high-speed implementations of block ciphers was done during the discussion phase of the AES competition [3] from 1997 to 2000. Only few implementations of thoroughly investigated block cipher algorithms have been published. In 2004, Lázaro et al. [64] present a Serpent design capable of encrypting 42.8 Gbit/s on a Xilinx Virtex-2 XC2V2000 FPGA. Their architecture is fully unrolled with two pipeline stages for each round of Serpent. In order to increase the throughput, the encryption pipeline is clocked at double frequency compared to the key scheduler. Sugier [103] describes an implementation of Serpent that uses full outer-loop pipelining and a key scheduler in combinational logic which achieves a throughput of 19.7 Gbit/s targeting an Xilinx Spartan-3E FPGA. A design employing pipelining for the cipher rounds of Serpent as well as for the key scheduler that reaches a throughput of 17.5 Gbit/s is presented in the same paper.

In order to use block ciphers to provide authenticated encryption, special modes of operation have been developed. These include GCM-AES which was standardized by the NIST [78] and thus is used for many protocols such as TLS [95], SSH [54] and IPSec [25] Consequently GCM-AES received significant attention from the research community, and several implementations targeting FPGAs can be found in the literature. However, GCM-AES is widely seen as an unsatisfactory and brittle standard that comes with some disadvantages that can compromise security [83]. Thus, in 2014, a new competition called *CAESAR* (Competition for Authenticated Encryption: Security, Applicability, and Robustness) was launched by the international cryptographic research community [27]. Its main goal is to identify authenticated ciphers that offer advantages over AES-GCM and that are suitable for widespread adoption. The whole competition is based on a public evaluation. For the first round of the competition, launched in march 2014, 57 authenticated cipher candidates were submitted, some of which have already been withdrawn because of security flaws that could be identified. Some of the submissions like ASCON [35] or MORUS [109], are fully parallelizable and only use functions that are cheap to implement in hardware and could prove to be very fast. However, as the competition is in a very early stage it is not yet foreseeable which submissions will finally be approved and currently only few hardware implementations of proposals have been published. Among those published, none is targeting a throughput near 100 Gbit/s. So, still most high-throughput implementations of authenticated encryption schemes are still based on GCM-AES.

In 2009, Zhou et al. [113] presented a single-core GCM-AES design, which targets a Xilinx Virtex-5 FPGA and achieved a throughput of 41.5 Gbit/s based on the 128-bit version of AES. Henzen and Fichtner [49] showed that it is possible to break the 100 Gbit/s barrier on a Virtex-5. They made use of four fully unrolled AES cores for the encryption part and used four Karatsuba-Ofman (KO) multipliers in order to realize the authentication part. Their design reaches a throughput of 119.3 Gbit/s. Abdellatif, Chotin-Avot, and Mehrez [2] describe a GCM-AES design reaching a throughput of 102.4 Gbit/s on a Virtex-5 FPGA. Their implementation is based on four fully unrolled, pipelined AES cores for encryption and four parallel authentication cores. However, their design uses a fixed secret key which allows them to precalculate the round keys for AES and the key-constants required for the authentication part and to synthesize those into their design. This reduces the complexity for both, the encryption part as well as for the authentication part but requires to reprogram the FPGA in case a key change is needed. Using the same principle they also report a single core design capable of 30.9 Gbit/s on a Virtex-5 in [1].

When using GCM, the most complex operation during the computation of a message

digest is actually the multiplication in the binary finite-field $GF(2^{128})$, which is part of the universal hashing function called GHASH. Therefore, most of the effort in improving GCM implementations has been spent on speeding up this calculation. Wang et al. [106] presented a GHASH architecture based on four GHASH cores that achieved a throughput of 123.1 Gbit/s on a Virtex-5. In [29], Crenne et al. reached 238.1 Gbit/s by using 8 parallel finite-field multipliers, also targeting a Xilinx Virtex-5 FPGA.

To the best of our knowledge, no hardware architecture based on a block cipher other than AES and targeting a high-throughput AE implementation has been presented so far. Moreover, no AES design, which makes use of an operation mode different than GCM in order to achieve throughputs up to 100 Gbit/s, has been published to date.

1.2 Outline

The remainder of this thesis is organized as follows. Chapter 2 defines the main cryptographic goals. It also introduces a communication model from a cryptographic point of view and describes basic attacks that try to break the main cryptographic goals. Chapter 3 introduces the concepts of symmetric-key cryptography and public-key cryptography. Furthermore, basic cryptographic primitives used in classical cryptography such as block ciphers, secure hash functions and message authentication codes are explained. The chapter also includes an introduction to quantum cryptography and raises the key-distribution problem which is then elaborated and discussed in Chapter 4. In Chapter 5, two block ciphers relevant for this thesis, namely the AES cipher and the Serpent cipher are described in detail. Next, Chapter 6 introduces the concept of authenticated encryption and shows methods to achieve AE. In addition, the Galois Counter mode of operation (GCM) and the Offset CodeBook (OCB) mode of operation, which are two special block-cipher modes for AE relevant for this work are explained in detail. The contribution of this work is based on an existing, specially designed FPGA-based hardware platform and an existing authenticated encryption implementation. The whole system was designed to provide a future-proof secure high-speed communication platform that is based on a quantum key-distribution system. Thus, first, Chapter 7 gives an introduction to reconfigurable hardware and second, Chapter 8 gives an overview on the actual system's hardware platform. Chapter 9 then describes the existing AE implementation in detail. In order to identify promising alternatives to the existing AE engine, Chapter 10 first investigates the requirements of the AE engine and then discusses and analyzes different block ciphers and modes for authenticated encryption in regard to their suitability for the previously defined requirements. In the end of this chapter, one alternative block cipher and one alternative AE mode are selected for implementation. Next, in Chapter 11, the actual implementation of a parallel pipelined Serpent cipher which achieves an throughput of over 100 Gbit/s is presented in detail. The design of the alternative AE engine which is based on the Offset CodeBook mode is then illustrated in Chapter 12. Chapter 13 describes how the implemented system was tested an verified and implementation results and a comparison with related work is given in Chapter 14. Finally, Chapter 15 summarizes the results and draws conclusions.

Chapter 2

Cryptographic Goals

There has been a dramatic increase of communication systems in the 20$^{\text{th}}$ century. Today's information society heavily relies on cryptographic systems that protect information and communication by providing confidentiality, data integrity, entity authentication and data origin authentication.

This chapter, in Section 2.1 lists and explains the main cryptographic goals. Then Section 2.2 defines a common model of communication and describes the different aspects and goals of information security based on this model. Finally, Section 2.3 defines an attack model and shows how malicious adversaries try to attack the communication between valid entities.

2.1 The Major Cryptographic Goals

Following [71] the **major cryptographic goals**, from which all other cryptographic goals can be derived are defined as follows:

1. **Confidentiality** or *privacy* is a method to keep the information secret from all but those parties authorized to have it.

2. **Data integrity** is a method providing protection of unauthorized alteration of data. It allows to detect data manipulation such as deletion, insertion and substitution by unauthorized parties.

3. **Authentication** is a method related to identification. It applies to communicating entities as well as to information. Whenever two start communicating they should identify each other and the information delivered over the channel should be authenticated with regards to origin, data content, date of origin, etc. This is why the aspect of authentication in cryptography is often subdivided into two main classes: *entity authentication* and *data origin authentication*. Data integrity implies data origin authentication, because a modification of the data also means that the origin of data has changed.

4. **Non-repudiation** is a method that prevents entities from disavowing previous commitments or actions.

2.2 Communication Model

In order to show how the cryptographic goals aim to achieve communication security a communication model needs to be defined. Using a standard model of communication helps to identify security problems that arise in practice. Section 2.3 will later clarify how this model can be violated with respect to a particular cryptographic goal.

Following [99] a *standard model of communication* can be defined as follows:

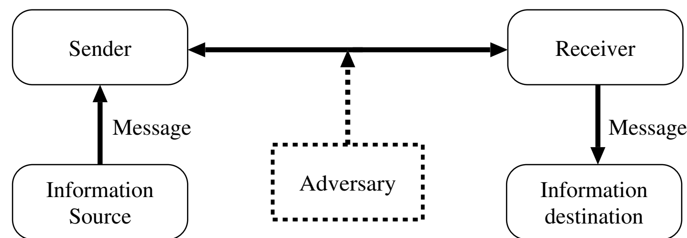


Figure 2.1: Standard communication model.

The communication model is divided into five main parts:

1. An *information source* which produces the message to be transmitted.
2. A *transmitter* or *sender* which processes the message such that it can be transmitted over the channel.
3. A *communication channel* is the medium used to transmit the message from transmitter to receiver. In a general case this channel is *insecure* and accessible to a third party, called *attacker* or *adversary* which can read, alter, delete and add messages. In contrast, a channel is said to be *secure* if only the two communication parties are able to read, alter, delete or add information transmitted over the channel. One method to make a channel secure is to make it physically inaccessible. However it is a very difficult task to achieve this *physical security.*
4. *Receiver(s)* which receives the message and which performs the inverse actions done by the transmitter to reconstruct the message.
5. A *destination* is the entity the message is addressed to.

All the parties that communicate over different channels together form a communication network. When the message is received correctly by the receiver as intended by the sender we speak of a normal communication flow. In reality however, many factors influence the communication flow and a correct flow is hard to achieve. On the one hand, there are non-malicious interferences on the channel. These interferences do not occur by intent and can be caused by technical or accidental problems. On the other hand, malicious interferences exist that are caused by intentional acts of an adversary.

As previously mentioned physical security is hard to realize. Thus, in order to prevent malicious attacks cryptography offers a set of primitives and methods to secure the channel.

2.3 Attack Model

Attacks, as previously mentioned are malicious actions performed by an unauthorized third party called *attacker* or *adversary* which aim to get an advantage out of the information transmitted over the channel by the legitimate communication partners.

There are three general types of attacks that are threats to the major cryptographic goals and that violate Shannon's model of communication [99, 100]. Using these attack schemes an adversary attempts to violate the goal of channel availability and the cryptographic goals of confidentiality, data integrity, authentication and non-repudiation.

1. *Attack on* ***availability:*** The adversary actively disturbs the communication channel and attacks the communication flow intended by the sender. The attack prevents the messages sent by the sender to be received by the intended receiver.

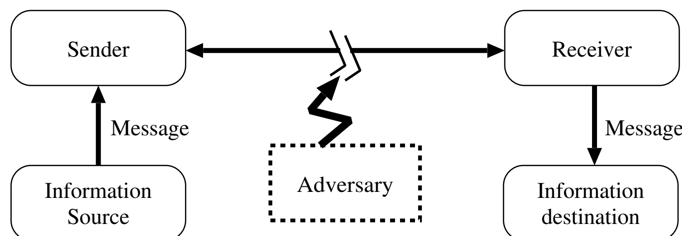


Figure 2.2: Attack on availability.

2. *Attack on* ***confidentiality:*** The adversary eavesdrops the communication and reads the messages sent between the communication partners and such violates their privacy.

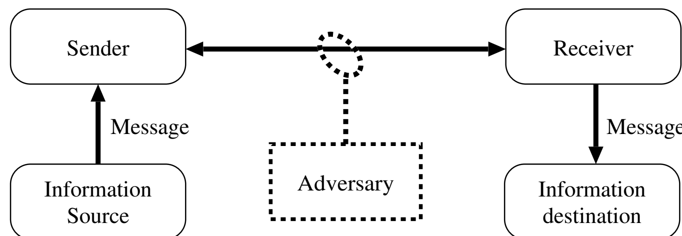


Figure 2.3: Attack on confidentiality.

3. *Attack on* ***authenticity:*** The adversary alters the normal communication flow. In

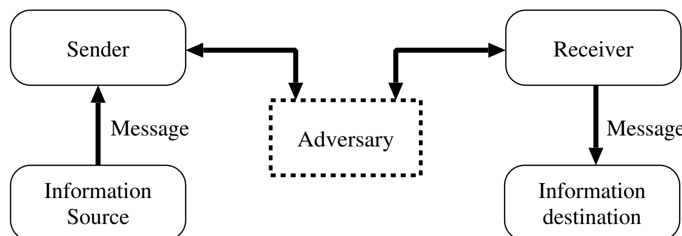


Figure 2.4: Attack on authenticity.

an *impersonation attack* the adversary sends messages to the receiver and pretends to be a valid and intended sender. If the adversary intercepts a message, modifies its

content and sends it to the receiver intended by the legitimate sender, she performs a *substitution attack*. In an *deception attack* the adversary can either impersonate another entity, replay messages intercepted from a valid entity, or actively intervene in protocol executions by selectively combining multiple parallel protocol executions. In all three cases the authenticity of the sender is attacked.

Depending on the adversary's behavior and its role, all types of attacks can be divided into two groups:

1. *Passive attacks:* The adversary eavesdrops the communication between the legitimate communications partners. By doing so the confidentiality of the data is being attacked.

2. *Active attacks:* The adversary attempts to alter, delete and add data. By actively changing the normal communication flow the attacker threatens the integrity and/or the authenticity of the data.

Cryptography tries to treat and prevent these attacks in theory and in practice. However, multiple methods exist to realize the different cryptographic goals. The following chapter introduces some of the primitives that cryptography provides.

Chapter 3

Cryptographic Primitives

In order to prevent an adversary from successfully breaking the cryptographic goals cryptographic primitives are put into place. The aim of this chapter is to introduce important methods and to show how these methods can be used in order to secure a public channel. All the primitives presented in this chapter are based on the Kerckhoff's Principle [57] which states, that the security of a cryptographic system should solely rely on the use of a secret key but not on the use of secret primitives. Section 3.1 deals with symmetric and asymmetric encryption that can be used to provide confidentiality of data. Methods to ensure data origin authentication are treated in Section 3.2 and Section 3.3 explains the ideas and principles of quantum cryptography.

3.1 Encryption Schemes

A *cipher* or *encryption scheme* is a primitive that aims at providing the goal of confidentiality. In order to do so an encryption scheme *enciphers* a plaintext P, using a key K, to give a resulting message C called ciphertext. This transformation is called encryption. The inverse of this transformation, named decryption, *deciphers* a ciphertext C using the same or a different key $\hat{K}$, resulting in the original plaintext P again.

Encryption schemes have a long history. Already some thousand years ago cryptography was used to protect information. Over the centuries many different cryptographic methods have been developed, that now are categorized into classical and modern (or newer) ciphers. Classical ciphers typically encrypt at the level of letters using substitutions, where a plaintext letter is substituted with a ciphertext letter, or using transpositions where the plaintext symbols are reordered following given rules in order to give the ciphertext. However, modern ciphers usually work on bit level and can be categorized into *symmetric ciphers* and *asymmetric ciphers*.

Symmetric encryption schemes use identical keys for both communicating partners, that are kept secret and that are used for encryption and decryption. In contrast, in *asymmetric encryption schemes* (also known as *public-key encryption schemes*) each communication party owns two different keys. One key, called public key, is provided publicly and used for encryption whereas the second key, which is called private key and which is kept secret by its owner, is used for decryption. To be secure, the public key strictly needs to be authenticated.

In the following we formalize the definition of public-key encryption schemes and

symmetric encryption schemes and show the existing variants.

3.1.1 Public-Key Encryption Schemes

Public-key cryptography uses a pair of associated keys $\{K_e, K_d\}$ instead of a single secret key. K_e is called the public key and K_d is called the private key. It has to be infeasible to determine the K_d when knowing K_e. $E_{K_e}()$ is the one-way function used for encryption and $D_{K_e}()$ is the corresponding inverse one-way function that can be used for decryption.

When the two communication partners Alice and Bob want to communicate, Bob creates a key-pair $\{K_e, K_d\}$ and sends his public key K_e to Alice but keeps his private key K_d secret. It is important, that the channel used to transmit the public key has to be authenticated. Alice can then encrypt a message P and send it to Bob using Bob's public key K_e and by applying the transformation $C = E_{K_e}(P)$. Bob can then decrypt this message using his private key by applying $P = D_{K_d}(C)$.

Public-key cryptography is able to fulfill all cryptographic goals defined in Section 2.1. In order to do so it only requires an authenticated channel. The authenticity of the channel is important as otherwise an adversary can distribute wrong public keys, for which she knows the corresponding private keys. In such case, the encrypted message would not be private anymore.

Public-key cryptography is much more complex than symmetric-key cryptography. Thus, in general, public-key cryptography is usually used to agree on secret keys that are then used for symmetric encryption schemes.

3.1.2 Symmetric Encryption Schemes

Following [14] a symmetric encryption scheme can be defined as follows:

Definition 1 (Symmetric Encryption Scheme)**.** A symmetric encryption scheme, $\mathcal{SE}$, consists of three algorithms ($\mathcal{K}$,$\mathcal{E}$,$\mathcal{D}$), as follows:

1. $\mathcal{K}$ is a randomized *key-generation* algorithm that returns a string K. If $\mathsf{Keys}(\mathcal{SE})$ is the set of all strings that have non-zero probability of being output by $\mathcal{K}$, then the members of this set are called *keys*. $K \xleftarrow{\$} \mathcal{K}$ denotes the operation of executing $\mathcal{K}$ to get the random key K returned.

2. The *encryption* algorithm $\mathcal{E}$ can be stateful or randomized. $\mathcal{E}$ takes a key $K \in \mathsf{Keys}(\mathcal{SE})$ and a *plaintext* $P \in \{0,1\}^*$ to return some *ciphertext* $C \in \{0,1\}^*$. $C \xleftarrow{\$} \mathcal{E}_K(P)$ denotes the operation of $\mathcal{E}$ on K and P.

3. The *decryption* algorithm $\mathcal{D}$ takes a key $K \in \mathsf{Keys}(\mathcal{SE})$ and a ciphertext $C \in \{0,1\}^*$ to return some string $P \in \{0,1\}^*$, such that $P \xleftarrow{\$} \mathcal{D}_K(C)$. It is required that $\mathcal{D}_K(C)$ returns P for each $C \xleftarrow{\$} \mathcal{E}_K(P)$ for any key $K \in \mathsf{Keys}(\mathcal{SE})$ and any message $M \in \{0,1\}^*$.

As the definition indicates, the key-generation algorithm $\mathcal{K}$ is randomized. It takes no input but flips coins internally and uses these to select a key K. Usually the key is a random string of a fixed length that is called the key length of a scheme. Two parties that want to use a scheme need to generate a key K using $\mathcal{K}$. Once both parties share K, the sender is able to run the encryption operation $\mathcal{E}$ with key K and plaintext P to produce the ciphertext C which is then transmitted to the receiver.

The encryption algorithm may either be randomized or stateful. When *randomized* the algorithm, each time it is called, flips coins internally and uses those to compute the output for a given pair K, P. Consequently, performing the operation $C_i = \mathcal{E}_K(P)$ twice may result in different results C_i. A *stateful* encryption algorithm in contrast, depends on a state that is initialized in a algorithm-specified an well-defined way. Whenever invoked, the algorithm computes the ciphertext C based on P, K and the current state. Finally, it updates and stores the state. There is no need for synchronization (apart from an initial synchronization) between the receiver and the sender. An encryption scheme that does not maintain a state is called *stateless*.

The receiver, upon receiving a ciphertext C computes the original message using $P = \mathcal{D}_K(C)$ using the same key used to generate C. The decryption algorithm itself is never randomized or stateful but deterministic. Depending on the allowed lengths of the input message P, symmetric encryption schemes are classified as *block ciphers* or *stream ciphers*.

Block Ciphers

Block ciphers can either be symmetric-key or public-key. However, in the following only symmetric-key block ciphers are treated.

Block ciphers are set of Boolean permutations operating on n-bit vectors [32] V_n of fixed size, where n denotes the *block length* of the cipher, such that the cipher maps n input bits to n output bits under the use of a k-bit cipher key K that takes values out of the key-space $Keys$.

Definition 2 (Block Cipher)**.** An n-bit block cipher is a function $\mathcal{E} : V_n \times \mathsf{Keys}(\mathcal{SE}) \rightarrow V_n$, such that for all keys $K \in \mathsf{Keys}(\mathcal{SE})$, the encryption function $\mathcal{E}(P, K)$ is an invertible mapping from V_n to V_n written as $\mathcal{E}_K(P)$. The decryption function which is denoted as $\mathcal{D}_K(C)$ is the inverse mapping of $\mathcal{E}_K(P)$ such that $P = \mathcal{D}_K(C)$.

Block ciphers operate on blocks of length n. In order to process messages exceeding the length of one block n those messages are processed by *modes of operation* which are cryptographic primitives based on block ciphers. Using these modes of operation, block ciphers can be used to accomplish a wide variety of cryptographic tasks. An introduction to some concrete modes of operation follows in Chapter 5.

Stream Ciphers

Stream ciphers are another class of symmetric encryption algorithms. While block ciphers typically operate on larger blocks of multiple bits of data, stream ciphers usually operate on smaller sets of data. Additionally, stream ciphers use transformations that vary over time. So, in contrast to block ciphers, the encryption result C_i not only depends on the secret key K and the plaintext message P_i but also on the current state σ_i of the encryption engine. Typically stream ciphers are fast and less complex to realize in hardware. Furthermore, as they have a limited error propagation they might be advantageous for systems where transmission errors are likely to occur. However, one thing to mention is, that no stream cipher that is standardized exists by today. Still, by using special modes of operation block ciphers can be turned into a stream cipher.

Typically stream ciphers are defined by two functions, f and g such that,

$$\begin{aligned}\sigma_{i+1} &= f(\sigma_i, P_i, K_i)\\ C_i &= g(\sigma_i, P_i, K_i)\end{aligned}$$

where f is the function to compute the next state σ_{i+1} and g is the actual encryption function. P_i is the bit (or number of bits) of the plaintext message P currently being processed and K_i is the corresponding key stream and σ_i is the present state.

In general, stream ciphers can be divided into two classes: *synchronous stream ciphers* and *asynchronous stream ciphers.*

Synchronous stream ciphers generate the key stream independently of the plaintext P and the cipher text C. So prior to communication the sender and the receiver have to be synchronized, to operate under the same key K and the same internal state σ_i. Synchronous stream cipher can be characterized by:

$$\begin{aligned}\sigma_{i+1} &= f(\sigma_i, K_i)\\ Z_i &= g(\sigma_i, K_i)\\ C_i &= h(Z_i, P_i)\end{aligned}$$

where f is the function to compute the next state and g is the function to produce the keystream and h is the output function that combines the keystream and the input message.

An *asynchronous stream cipher* is a stream cipher that generates the keystream as a function of the key and a fixed number of previous ciphertext bits:

$$\begin{aligned}\sigma_{i+1} &= f(C_{i-t}, C_{i-t+1}, \ldots, C_{i-1})\\ Z_i &= g(\sigma_i, K_i)\\ C_i &= h(Z_i, P_i)\end{aligned}$$

where the next state σ_{i+1} is ciphertext dependent and where an initial state σ_0 exists. As the state—and thus the keystream—is only dependent on a few previous ciphertext bits, the cipher is self-synchronizing even if some ciphertext bits are transmitted erroneous.

3.1.3 Symmetric versus Asymmetric Encryption

When comparing symmetric encryption algorithms and asymmetric encryption algorithms the following advantages and disadvantages can be found:

- Symmetric encryption schemes typically execute much faster in software as well as in hardware than asymmetric encryption schemes with a comparable security level.
- Symmetric encryption scheme implementations usually are less resource consuming in terms of required area, power, energy, memory, etc.
- Symmetric encryption schemes require both parties to share a secret key. Distributing the secret key over a (secure) channel is known as the *Key Distribution Problem* and comes at some cost. To provide a high level of security, this shared secret key often needs to be changed and kept secure. The procedure of generating, managing, redistributing, storing and adopting the key in a secure way often is a complicated task to accomplish.

Increasing speeds of networks clearly favors fast encryption schemes. Thus, in this work we rely on symmetric encryption schemes.

3.2 Authentication Schemes

Data integrity is closely related to data origin authentication. In case a message is altered its integrity has been attacked and consequently the sender is not authenticated any more. Thus, data integrity and data origin authentication are intrinsically tied together. Consequently, when one of them is compromised also the other one has to be doubted. Hence, mechanisms for data authentication have to provide data origin authentication and data integrity.

An important class of cryptographic primitives used for authenticity is called *hash functions*. Hash functions in general can be divided into *unkeyed hash functions* which are also called *modification detection codes (MDCs)* and *keyed hash functions* also known as *message authentication codes (MACs)*.

Definition 3 (Hash Function)**.** A hash function h (in its most unrestricted definition) has to provide to properties:

1. The function h has to provide *compression* by mapping an input x of arbitrary (finite) bitlength to an output $y = h(x)$ of fixed bitlength n.

2. Given h and an input x the output $y = h(x)$ has to be *easy to compute*.

The basic idea of hash functions is that the hash tag serves as a small-sized representative for the input string and can be used as a unique identifier for that string and can thus warrant the integrity of the string. However, as a hash function maps messages of arbitrary finite length of a domain D to an output of n-bit length in range R where $|D| > |R|$, there have to exist collisions where two distinct input messages x and x' map to the same output $y = h(x) = h(x')$.

In the remainder of this section we will first describe unkeyed hash functions and then show how to transform these into keyed hash functions to create concrete authenticity primitives.

3.2.1 Unkeyed Hash Functions

An unkeyed hash function, also called modification detection code (MDC) takes an input of arbitrary length and compresses it to a fixed length output called *hash value* or *hash tag*. As per definition, the space of possible outputs $y = h(x)$ is restricted to 2^n, where n is the bitlength of the hash tag, whereas the space of the input x is unlimited (but still finite). An cryptographic hash function, in addition to the properties given in Definition 3 has to fulfill the following properties [71]:

1. ***Preimage resistance:*** It should be computationally infeasible to find a preimage x' hashing to a given prespecified hash value y such that $y = h(x')$.

2. ***Second preimage resistance:*** It should be computationally infeasible to find any input x' that hashes to the same output as a given input x. So it should be hard to find any input $x' \neq x$ hashing to $y = h(x') = h(x)$. Another term for second preimage collision resistance is *weak collision resistance*.

3. ***Collision resistance:*** It should be computationally infeasible to find any two inputs x and x' hashing to the same value $y = h(x) = h(x')$. Collision resistance is also called *strong collision resistance.*

In addition, for cryptographic MDCs the following properties are of interest:

1. ***Non-correlation:*** Input bits and output bits should not correlate and it is desirable to have an avalanche effect where every input bit effects every output bit.
2. ***Near-collision resistance:*** It should be hard to find any two input messages x and x' such that the output of $h(x)$ and $h(x')$ differ only in a small number of bits.
3. ***Partial preimage resistance*** or ***Local one-wayness:*** It should be hard to recover any substrings or even the entire input message even if parts of the input message are known.

3.2.2 Keyed Hash Functions

Hash functions that involve the use of a secret key are called *keyed hash function* or *message authentication codes (MACs).* MACs, in contrast to MDCs, can also be used to provide data origin authentication and data integrity.

Definition 4 (Message Authentication Code (MAC))**.** A message authentication code is a class of functions h_k that is parametrized by a secret key k and that is characterized by the following properties [71]:

1. For a function h_k and a given input message x and a given key k the output of $h_k(x)$ has to be *easy to compute.* The resulting output is called *MAC-value* or *MAC.*
2. The function h_k has to provide *compression* by mapping an input x of arbitrary (finite) bitlength to an output $h_k(x)$ of fixed bitlength n.

   In addition, for a given function h and for all fixed allowable (but unknown to the adversary) key values k, the following properties have to hold:
3. It has to provide *computation resistance* by making it infeasible to compute any text-MAC pairs $(x, h_k(x))$ for any new input $x_i \neq x$ (including possibly $h_k(x) = h_k(x_i)$ for some i), given zero or more text-MAC pairs $(x_i, h_k(x_i))$.

MACs can be constructed using different approaches. Based on their underlying compression function they can be classified into the following groups:

- ***Based on block ciphers:*** Many MAC algorithms used are based on block ciphers and often use these in cipher-block-chaining mode (CBC) [76] or electronic codebook mode (ECB) [76].
- ***Based on hash functions:*** These constructions use MDCs (hash functions) in combination with a secret key that is part of the input. An example for this type of construction is the HMAC [60, 75] construction with SHA-3 [82] as underlying MDC.
- ***Customized MACs:*** Those MAC algorithms are specially designed ("from scratch") for the purpose of message authentication. Often these algorithms exploit the properties of hash function families and are called universal hash based MACs.

3.3 Quantum Cryptography

Quantum cryptography (QC) is not a classical cryptographic primitive but has been topic of extensive research in recent years and has clearly evolved to a practical level. The idea of QC was first proposed by Wiesner [107] (1983) and by Charles H. Bennett and Gilles Brassard [18] (1984). It is based on the existence of indivisible quanta and entangled systems and some of the fundamental principles of quantum mechanics [47]:

1. Every measurement perturbs the measured system.
2. According to the Heisenberg uncertainty principle it is impossible to simultaneously determine the position and the momentum of a particle precisely.
3. It is impossible to determine the polarization of a photon in vertical-horizontal basis simultaneously to the diagonal basis.
4. It is impossible to draw pictures of individual quantum processes.
5. It is impossible to copy or duplicate an unknown quantum state.

These negative characteristics of quantum mechanics turn out to be useful in QC. As mentioned, a principle of quantum mechanics is:

Every measurement perturbs the system.[1] (3.1)

This fact can be used to detect eavesdropping when used in QC. Imagine, Alice codes information into single photons she sends to Bob, then Bob can determine eavesdropping by checking that he receives the photons unperturbed. If an evil attacker Eve wants to catch information on the photons she has to do measurements and consequently perturbs the system. To check whether someone listened to their communication all Alice and Bob have to do is to compare a random subset of the transmitted data using a public channel. If this subset is equal there was no eavesdropping.

So Alice and Bob can detect eavesdropping but only after having transmitted a message. This is not practicable when secret information has to be exchanged. Thus, in order to ensure privacy in advance Alice and Bob only exchange a random secret key over the quantum channel and use this key to later encrypt the actual secret data typically over a classical channel. If they realize eavesdropping while exchanging the secret key they simply discard it and exchange a new one. Consequently they can assure that no secret information of value is lost.

In order to ensure that Axiom (3.1) applies the data has to be encoded into non-orthogonal states. In practice individual photons that are either sent over optical fibres or over free space are used to encode the quanta or qubits (for quantum bits). For example the horizontal and the vertical polarization of a photon could be used to code the bit values of bit 0 and bit 1. As a second and diagonal base the $\pm 45°$ linear polarization with $+45°$ for bit 1 and $-45°$ for bit 0 could be used. Alternatively, the circular polarization basis could be used as a second base.

Section 3.3.1 will show a practical example that make use of the principles described above.

[1] Except if the quantum state is compatible with the measurement.

3.3.1 Example of the BB84 Protocol

The BB84 protocol was proposed in 1984 by Bennet and Brassard in [18]. In its original description the protocol uses photon polarization states to transmit the information. However any two conjugate pairs of states can be used to implement the protocol. The following description of BB84 will stick with the original proposal.

Principle

The protocol uses four quantum states that can be represented as up $|\uparrow\rangle$, down $|\downarrow\rangle$, left $|\leftarrow\rangle$ and right $|\rightarrow\rangle$ which constitute two bases. The two bases conjugate as all vectors of one base have equal-length projections in the vectors of the other base, e.g. $|\langle\uparrow | \leftarrow\rangle|^2 = \frac{1}{2}$. By convention the two states $|\uparrow\rangle$ and $|\rightarrow\rangle$ represent a binary 0 and the states $|\downarrow\rangle$ and $|\leftarrow\rangle$ denote a binary 1. The four spin states can also be represented as polarization states "horizontal" and "vertical" (belonging to the base $+$) and "$+45°$" and "$-45°$" (belonging to the base $\times$).

First, Alice sends random individual spins which represent random bits to Bob over a quantum channel that allows a one-to-one communication between the two parties. Next, Bob chooses a random base for each incoming qubit and measures its spin according to the chosen base. So, whenever Alice and Bob chose the same base they get correlated results but when they pick different bases they get uncorrelated results. Thus, on average Bob receives a string of bits with an error rate of 25%, called the *raw key*.

Table 3.1: Example key agreement using the BB84 protocol.

Alice's random bit	0	1	1	0	1	0	0	1
Alice's random sending basis	$+$	$+$	$\times$	$+$	$\times$	$\times$	$\times$	$+$
Photon polarization Alice sends	$\uparrow$	$\rightarrow$	$\searrow$	$\uparrow$	$\searrow$	$\nearrow$	$\nearrow$	$\rightarrow$
Bob's random measuring basis	$+$	$\times$	$\times$	$\times$	$+$	$\times$	$+$	$+$
Photon polarization Bob measures	$\uparrow$	rand	$\searrow$	rand	rand	$\nearrow$	rand	$\rightarrow$
Public discussion of basis	OK	drop	OK	drop	drop	OK	drop	OK
Shared sifted key	0		1			0		1

In order to perform an error correction Bob announces the base he has chosen for measuring each corresponding qubit over a public channel. Alice then publicly tells Bob whether the base he has chosen is compatible with the state she has chosen to encode the qubit. In case the states is compatible they keep the bit and otherwise they discard it. So, around 50% of the bit string is discarded which results in a shorter key called *sifted key*. Note, that the error correction is performed on a public channel that Eve can listen to. To prevent Eve from modifying the information transmitted in this process this channel has to be authentic (but not necessarily confidential).

Attack Scenarios

If Eve intercepts a qubit, Bob will simply inform Alice that he didn't receive it as expected and tell her to discard it. Thus, Eve has no information gain and can only lower the transmission rate. So, for real eavesdropping Eve needs to send a new qubit (that ideally would be a copy of the intercepted one) to Bob after intercepting the qubit sent by Alice.

However, it can be proved that perfect copying is impossible in the quantum world [72, 108]. A simple and practical eavesdropping strategy that Eve could use is called intercept-resend. Eve measures each qubit in one of the two bases, and then transmits another qubit to Bob that correlates to the state she measured. As Eve does not know the base Alice chose she will be lucky in about 50% and resend a qubit with the correct base to Bob such that the legitimate communication partners will not realize her intervention. However, in the other halve of the tries she will be wrong and use a basis incompatible with Alice's and Alice and Bob will notice the intrusion. This clearly emphasizes the importance that Alice chooses the basis truly random. Using the intercept-resend strategy Eve can get about 50% information gain, while Alice and Bob will have about 25% errors in their sifted key which would definitely reveal the presence of Eve. In case Eve only applies the strategy in 10% of the cases, this will give her an information gain of around 5% while Alice and Bob will have an error rate of around 2.5% in their sifted key. The next subsection will show how such attacks can be countered.

Error Correction, Privacy Amplification

The sifted key Alice and Bob have been sharing so far contains errors due to technical imperfections and eventually due to the intervention of Eve. With today's technologies typical error rates lie around a few percent (which is beyond comparison with typical optical communication with error rates of about 10^{-9}). So in the first step called *information reconciliation*, Alice and Bob use classical error correction algorithms in order to get rid of the errors. Then, to lower Eve's information on the final key, they use a strategy called *privacy amplification*. In practice Alice and Bob often use a cascading protocol and compare the parity of blocks of the sifted key. In case of an error a binary search is performed to find and correct the error as proposed in [23]. This process is performed recursively until all blocks have been compared. Then the key stream is reordered and the procedure is repeated for several rounds. In the end Alice and Bob have identical keys with high probability. However, Eve has gained additional information on the key because of the parity bits exchanged. Privacy amplification is then used to lower Eve's information on the key to a arbitrarily low value. This could be done using a hash function that takes a string of length equal to the key and outputs a final key string of chosen shorter length.

3.3.2 Other Protocols

A variety of protocols for QC using either non-orthogonal or entangled quantum states have been published. In the following, a small selection of these will be presented. First, the *Two-State Protocol* and the *Six-State Protocol* which are both based on non-orthogonal quantum states will be described and then the *EPR Protocol* which uses entangled quantum states will be introduced. For a more complete overview of protocols please refer to [36, 47].

Two-State Protocol

The two-state or B92 protocol by Bennet [16] only uses two non-orthogonal states. Alice randomly sends one of the two states and Bob then performs projections onto subspaces orthogonal to the signal states. In case Bob chooses a base non-orthogonal to the one Alice used he will obtain a result in 50% of the cases whereas he will obtain an inconclusive outcome if he chooses a base orthogonal to the one the qubit was encoded with. After

transmission Bob tells Alice when he detected a bit but he does not announce the base he used because the base uniquely identifies the bit Alice sent.

However, in practice this system is only secure in lossless systems because while it is true that two non-orthogonal states cannot be distinguished precisely without perturbation it can be done at the cost of some losses. Eve could intercept the qubits and perform measurements. If she obtains an inconclusive result she blocks the qubit while she retransmits a copy in the correct state to Bob if she detects a state (because she then knows the state with certainty). To disguise the photons she blocked, Eve could send a pulse of higher intensity, such that Bob cannot realize a decrease in transmission rate.

One way to counter this attack is to encode the bits into phase differences between a strong reference pulse and a dim pulse (with less than on photon on average). In this case Bob can monitor the bright pulses and make sure that Eve does not remove any of them. Furthermore Eve can not remove one of the dim pulses because the interference of the strong pulse with vacuum would introduce errors that would be detected.

Six-State Protocol

In the six-state protocol [11, 24] three non-orthogonal bases are used for encoding the qubits with six states. Accordingly, the probability that Alice and Bob choose the same base is only $\frac{1}{3}$. However, this disadvantage of compared to the BB84 protocol is outweighed by the fact that eavesdropping inducts higher error rates. In case Eve tries to perform the previously mentioned intercept-resend attack and measures every qubit, this results in an error rate of around 33%, compared to 25% when applied in the BB84 protocol.

EPR Protocol

The original EPR protocol is based on quantum entanglement and Bell's inequalities and was originally proposed by Ekert [37]. A simplified version that does not invoke Bell's inequalities was presented by Bennett, Brassard and Mermin [17].

The idea of the simplified version is to replace the quantum channel between Alice and Bob by a channel carrying two entangled qubits from a common source. Both communication parties Alice and Bob measure the qubits independently using random bases. They then compare their bases and only keep the bits in case of congruence. In fact this protocol is very much alike the BB84 protocol. The only difference is that Alice does not send particles but she measures her particle of the entangled pair within one of the two bases.

3.3.3 Security of Quantum Cryptography

The security of QC is based on basic quantum-mechanic principles. Classical QC protocols consist of two phases:

1. A physical device generates quantum mechanic signals that are distributed and measured. They are then transformed into classical data.

2. Alice and Bob use a classical channel to discuss their data and to perform error correction and privacy amplification.

The unconditional security of those phases can be proved for theoretical systems [68, 69, 101] and even for imperfect devices [48]. However, it should be noted that real world systems suffer from technical imperfections that allowed practical attacks on real word

implementation. For a good overview on the security of QC and attacks theoretic as well as practical attacks that were possible due to imperfect implementations please refer to [89, 97].

3.3.4 Summary and Discussion

As shown, QC does not provide a complete solution for all cryptographic goals but can only be used in conjunction with classical symmetrical cryptographic primitives. Thus, a more precise name for QC is *quantum key distribution* as this is the purpose of QC. In addition it should be noted once more, that QC requires the two communication partners to be authenticated. Consequently, Alice and Bob need to share a small secret prior to communication. QC can then provide them a longer one and parts of his longer key could then be used as a starting secret for the next session.

For a wider and more complete overview as well as detailed discussions on quantum cryptography, the security of QC, attacks and technical realizations please refer to [7, 36, 47].

3.4 Conclusion

Cryptographic primitives based on the Kerckhoff's Principle, rely on the use of secret keys. In order to provide confidentiality symmetric or asymmetric encryptions can be used. MACs can be used to provide data origin authentication and data integrity. The problem of distributing these secret keys required for these primitives among the communication partners is know as the *key-distribution problem.* One possible solution to this problem is the use of public-key cryptography and another solution is based on the use of quantum cryptography. However, more solutions to this problem exist. Chapter 4 will discuss these solutions in detail.

Chapter 4

Key-Distribution Problem

The security of modern cryptographic algorithms, based on Kerckhoff's Principle, does not rely on the obscurity of an algorithm but only on the use of a secret key. Thus, secure distribution of secret keys among legitimate users is a central problem in cryptography that is known as the *Key-Distribution Problem* or as the *Key-Establishment Problem*. Basically there are five cryptographic ways to solve the Key Establishment Problem [7]:

1. **Classical Information-theoretic schemes:** Cryptosystems are called information-theoretically secure if they derive their security solely from information theory and thus do not make any assumptions on the hardness of any mathematical problems. Consequently, such systems are still secure even if an adversary has unbounded computing power. A well known example for an information-theoretically secure scheme is the One-Time Pad (OTP). If two parties are in possession of correlated strings that feature more correlation than any string that an eavesdropper could possess, it is possible to establish an information-theoretically secure key over a classical channel using public discussion, as shown in [66]. In fact, as will be shown later in this section, the use of a quantum channel allows a practical realization to provide such correlated strings of information.

2. **Classical public-key cryptography:** Public-key cryptography or asymmetric encryption is based on the difficulty of mathematical problems for which no polynomial algorithm exists to solve them. Therefore, by using secret keys that are long enough the computing resources to find solutions to these problems become infeasible. Public-key cryptography relies on "provable computational security". However, it should be noted, that it is not unconditionally secure as the underlying mathematical problems are not unsolvable and as the non-polynomial complexity has not been proven yet.

   As previously mentioned in Section 3.1, asymmetric encryption schemes include a public key that is accessible to anyone in order to allow encryption and a private key that allows the legitimate recipient to decrypt the message. As shown by Diffie and Hellman in [34], public-key cryptography provides a very practical way to establish a secret key over a classical public channel using no previously shared secret. However, in order to provide authenticity of the distributed keys, it is necessary to introduce a trusted authority. These trusted authorities provide a public-key infrastructure (PKI) that issues certificates for the users' public keys. Today, this system is widely used in the internet in the absence of any other practical solution for key distribution.

As asymmetric encryption schemes are typically much slower than symmetric encryption schemes, usually public-key cryptography is only used to establish a shared secret key between to two parties that then use symmetric encryption schemes for data encryption and authentication.

3. **Classical computationally secure symmetric-key cryptographic schemes:** Before the invention of public-key cryptography in 1976, symmetric-key encryption schemes were the only way to encrypt data. In case a secret key has been shared between two parties by any means it is possible to encrypt a message to provide a secret key for the key distribution protocol. Hence, key distribution that uses symmetric-key encryption always relies on a pre-established symmetric secret needed to allow authentication. In a sense, symmetric-key based key distribution does *key expansion* more than *key distribution*.

   In [100] Shannon proved, that there exists no other unconditionally secure encryption scheme that uses less key material than the One-Time Pad which requires the key material to be as long as the data to be encrypted. Consequently it is impossible to build an unconditionally secure key expansion based on classical cryptography. This follows from the fact that classical messages in contrast to quantum messages can be copied. Still, it is possible to build key-distribution systems based on symmetric-key encryption and authentication schemes that are not unconditionally secure. The Diffie-Hellman Key-Exchange Protocol [34] is an example of a key-distribution based on symmetric encryption schemes. As previously stated, symmetric encryption is typically much faster and less computation intensive than asymmetric encryption.

4. **Quantum key distribution (QKD):**

   Quantum Key Distribution (QKD) is based on the laws of quantum mechanics and takes advantage of the fact that it is impossible to gain information about non-orthogonal quantum states without disturbing and influencing these states [84]. It has been proven to be unconditionally secure and can be used to establish a random key between to parties that is perfectly secret from an information-theoretic point of view as shown in [19, 66, 89].

   A QKD system typically consists of a classical public channel, a secure quantum channel and an optional service channel as shown in Figure 4.1. First, as outlined in Section 3.3, one of the two parties involved, e.g. Alice, generates a random stream of classical data and encodes them into a sequence of quantum states of light which is then transmitted over the quantum channel. Bob, the second party involved, measures the quantum states and generates a stream of classical data that correlates with Alice's bit stream. The classical public channel, or the service channel which is public as well, is then used to check the correlation between the two bit streams. In case the correlation is high enough, this implies that no significant eavesdropping by an attacker has taken place as eavesdropping would influence and change the quantum states. Finally, if the correlation was high enough, the step of information reconciliation where bit errors are removed and the step of privacy amplification where the knowledge an eavesdropper Eve could have gained on the key is eliminated by generating a new key out of the correlated data using for example an universal hash function. Both these steps are again performed over the public channel. Therefore, the quantum channel is used to agree on a perfectly secure symmetric key that can later be used to symmetrically encrypt and send the data over the public channel.

However, as QKD is a symmetric key distribution technique it is necessary to perform authentication between the two parties involved. Consequently, a small secret key has to be shared in advance between both parties.

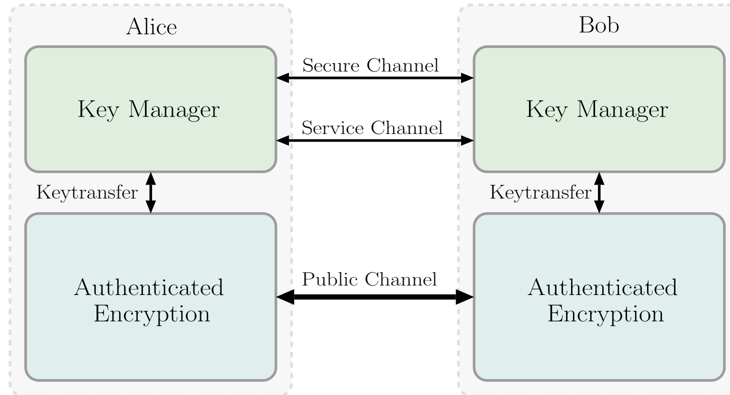


Figure 4.1: Overview of a quantum key distribution system.

Due to the fact, that today's quantum channels are much slower (typically between 1 kbit/s to 1 Mbit/s) than classical channels, the quantum channel is only used for key exchange and agreement and the classical channel is then used to transmit the actual data using symmetric encryption.

It should be noted that a QKD would also stay secure within the presence of quantum computers.

5. **Trusted couriers:** Trusted couriers are the oldest method to distribute secret keys. Even if it is still in use nowadays in some highly sensitive environments such as government intelligence or by banks to distribute the credit card PIN number to customers, it is highly unpractical in large systems and digital networks.

Chapter 5

Block Ciphers Used for the Thesis

In the early 1990s it became obvious that the security of the *Data Encryption Standard (DES)* [43] was not future proof any more. Therefore, in 1997, the National Institute of Standards and Technology (NIST) together with the industry and the cryptographic community started the search for a new standardized symmetric-key scheme. The goal was to find an unclassified, publicly disclosed and royalty-free encryption algorithm that should support a minimum block size of 128 bits and key sizes of 128, 192 and 256 bits. Therefore, in 1998 the First Advanced Encryption Standard Candidate Conference (AES1) was held and 15 potential block cipher candidates were presented and discussed. Followed by a public discussion phase, the Second Advanced Encryption Standard Candidate Conference (AES2) was held in 1999 and five candidates were presented that made it to the final round: *MARS* [26], *Rivest Cipher 6 (RC6)* [90], *Rijndael* [31], *Serpent* [10] and *Twofish* [98]. Those five candidates, also called AES finalists, were then extensively analyzed and explored and publicly commented in regard to their security, performance in software and hardware, intellectual property and many more properties. At the last conference, the AES3, in 2000 all submitters were given the chance to give their view on comments of their proposal. Finally, in October 2000, using all the information gained, NIST announced Rijndael as the winner of the competition [79].

In addition to the block ciphers proposed during the AES Candidate Conference lots of other block ciphers have been developed. However, most cryptographic research on the security and the performance has concentrated on the AES finalists. In the following, two AES finalists are described in more detail as they are of special interest for this thesis. First, in Section 5.1 the *Rijndael (AES)* block cipher will be described and second, in Section 5.2 the *Serpent* block cipher will be explained.

5.1 Rijndael (AES)

As previously mentioned, Rijndael was the winner of the AES competition and as a result it was standardized by NIST in the Federal Information Processing Standard (FIPS) 197 [80] that describes a specific version of Rijndael now commonly known as the Advanced Encryption Standard (AES) which restricts the block size to 128 bits and the key size to 128, 192 and 256 bits. Dependent on the used key size the algorithm is referred to as AES-128, AES-192 or AES-256.

The AES cipher belongs to the symmetric-key encryption schemes and is a block cipher

that encrypts independent input blocks of 128 bits of data. As the ciphers requires input blocks of exactly 128-bits size a padding functions fills up input messages to multiples of 128 bits if required. For encryption/decryption longer messages are then split up into blocks of 128 bits and processed iteratively or in parallel depending on the AES implementation.

The following Sections describe the AES algorithm in detail. First, Section 5.1.1 explains the data representation used in the AES algorithm. Second, the actual algorithm is described in Section 5.1.2 which covers the AES Round Transformation and in Section 5.1.3 that treats the round-key generation of AES. For an in-depth explanation of the cipher please refer to [32] and [80].

5.1.1 Data Representation

The smallest data entity used in AES is a byte. Each 128-bit input block, consisting out of bits $d_0 \dots d_{127}$, is split into sixteen bytes $b_0 \dots b_{15}$ where $b_0 = d_0 \dots d_7$ and where d_0 being the most significant bit and d_7 being the least significant bit of b_0 as shown in Figure 5.1.

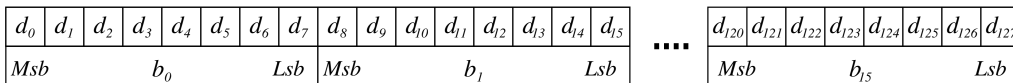


Figure 5.1: Bit mapping of an input block.

Internally, each input block is mapped to a 4×4 Matrix called *State* whose elements have an index s_{ij}, where i defines the row and j defines the column of the element. Figure 5.2 shows how the data blocks I_i of an input block are mapped to the state elements S_{ij} and how these are mapped to the output data blocks O_i. All operations of the AES algorithm operate on rows, columns or single elements of this state.

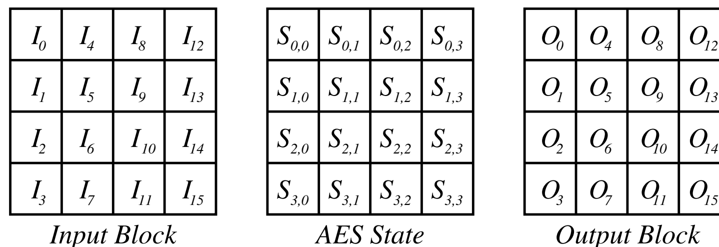


Figure 5.2: Mapping of an input block to the AES state and to an output block.

5.1.2 Round Transformation

The number of rounds N_r and the number of key columns N_k of the AES algorithm are dependent on the used key size. For a key size of 128 bits $N_r = 10$ and $N_k = 4$, for 192 bits $N_r = 12$ and $N_k = 6$ and for 256 bits $N_r = 14$ and $N_k = 8$. Except for the very last round that is slightly different, all rounds of AES encryption algorithm are defined the same using four basic operations: *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*. The AES decryption algorithm requires the inverse operations: *InvSubBytes*, *InvShiftRows* and *InvMixColumns* while the *AddRoundKey* stays the same. All these operations will be described in the following sections.

The initialization of the state matrix for the AES encryption starts by applying the initial *AddRoundKey* operation which adds (XORs) the secret user key to the input message. After

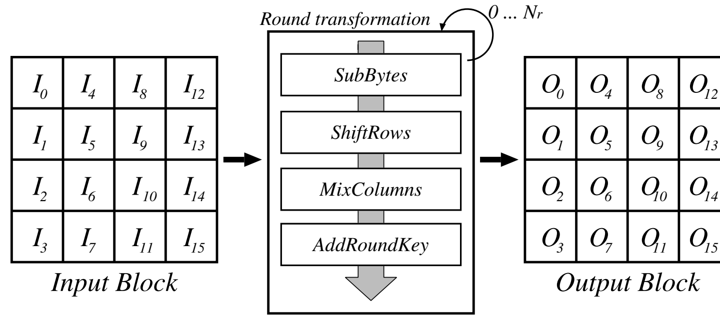


Figure 5.3: Simplified overview of the AES encryption.

this first step the round transformation iteratively executes the *SubBytes*, the *ShiftRows*, the *MixColumns* and the *AddRoundKey* operations. For the *AddRoundKey* operation an appropriate round key that is derived from the secret user key is required. These round keys can be computed as described in Section 5.1.3. The last round slightly differs as the *MixColumns* operation is omitted in this round. The final state matrix then constitutes the encrypted message. Figure 5.3 gives a simplified overview of the AES encryption process.

The AES decryption is the inverse operation of the AES encryption. First, the initial *AddRoundKey* operation is performed using the last round key instead of the user key. Second, instead of the operations used for encryption their inverse counterparts *InvSubBytes*, *InvShiftRows* and *InvMixColumns* have to be used. In addition, the order of the operations differs as it starts with *InvShiftRows*, followed by *InvSubBytes*, *AddRoundKey* and ends with *InvMixColumns*. The *AddRoundKey* operation stays the same but uses the round keys in reversed order. Hence, for round i round key number $N_r - i$ is required. Consequently, the on-the-fly computation of the round keys is slower compared to the AES encryption. Similar to the AES encryption the *InvMixColumns* operation is omitted in the last round. The final state matrix then constitutes the decrypted message.

SubBytes/InvSubBytes

The *SubBytes* operation is a non-linear transformation that substitutes each byte of the input state according to a substitution table called S-box that is derived by first taking the multiplicative inverse of the finite field $GF(2^8)$ for each byte and then applying an affine transformation over $GF(2)$ as described in [80].

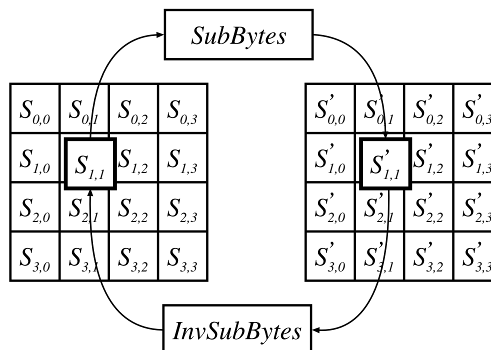


Figure 5.4: The *SubBytes* transformation and its inverse the *InvSubBytes* transformation.

In order to perform the *InvSubBytes* operation first the inverse affine transformation in $GF(2)$ is applied followed by taking the multiplicative inverse in $GF(2^8)$ for each input byte. Both these substitutions can either be realized using a precomputed S-box table or by an on-the-fly computation. For a more detailed description see [32].

ShiftRows/InvShiftRows

The *ShiftRows* and the *InvShiftRows* operations rotate each row of the state by n-bytes, where n is the index of the row. The only difference is that the rotation for *ShiftRows* transformation is a left rotate but a right rotate for the *InvShiftRows* transformation. Figure 5.5a and Figure 5.5b show the effects of the *ShiftRows* and the *InvShiftRows* operation on the state matrix.

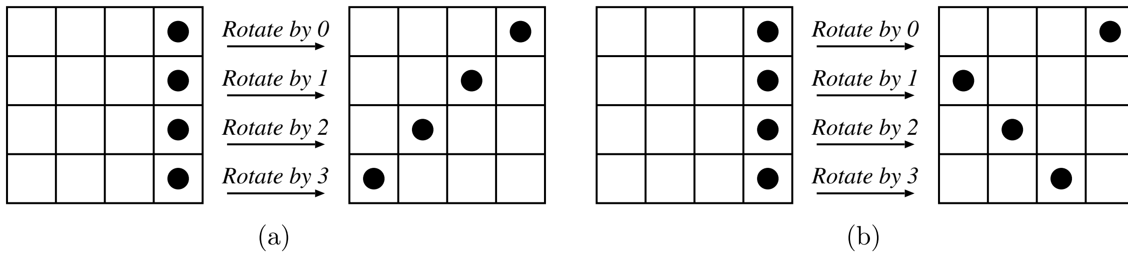


Figure 5.5: The effect of the *ShiftRows* transformation (a) and its inverse the *InvShiftRows* transformation (b) on the state matrix.

MixColumns/InvMixColumns

The *MixColumns* and the *InvMixColumns* transformations can be described as matrix multiplications that operate on columns of the state matrix. A detailed notation of the matrix multiplication performed in the *MixColumns* operation is given in Equation 5.1 and Equation 5.2 describes the equivalent for the *InvMixColumns* operation.

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \tag{5.1}$$

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \tag{5.2}$$

AddRoundKey

During the *AddRoundKey* operation each byte of the state matrix is XORed with the corresponding byte of the round key. As previously stated, the round key can either be computed on-the-fly or be precomputed and stored in memory. Details on the round-key generation are given in Section 5.1.3.

5.1.3 Round-Key Generation

As previously mentioned, for each round of the AES algorithm one 128-bit round key is required. All these round keys are derived from the secret user key using the *KeyExpansion* algorithm as described in Algorithm 5.1.1. Depending on the user key size $(N_r + 1) \cdot 4$ columns of 4 bytes are generated where the first N_k columns are those provided by the user key. The *SubWord()* function as used in Lines 12 and 14 takes a word of four bytes and applies the *SubBytes* operation for each byte. The *RotWord()* function in Line 12 rotates a word of four bytes to the left by one byte and *Rcon* is an array holding one constant for each round.

Algorithm 5.1.1 AES key-expansion algorithm.

Input: User key K of N_k words length.
Output: $KeyExpansion(K, N_k)$

1: word $temp$
2: $i = 0$
3: word $roundkeys[4 \cdot (Nr + 1)]$
4: **while** $i < N_k$ **do**
5: $roundkeys[i] = K[i]$
6: $i = i + 1$
7: **end while**
8: $i = N_k$
9: **while** $i < 4 \cdot (N_r + 1)$ **do**
10: $temp = roundkeys[i - 1]$
11: **if** $i \mod N_k = 0$ **then**
12: $temp = SubWord(RotWord(temp)) \oplus Rcon[i/N_k]$
13: **else if** $N_k > 6$ and $i \mod N_k = 4$ **then**
14: $temp = SubWord(temp)$
15: **end if**
16: $roundkeys[i] = roundkeys[i - N_k] \oplus temp$
17: $i = i + 1$
18: **end while**
19: **return** $roundkeys$

Note, that the AES decryption requires the round keys to be applied in reverse order. This is a drawback because it increases the computation time as all round keys have to be computed prior to the first round of AES.

For a detailed description of the round-key generation and the calculation of the constants held by the *Rcon* array please refer to [80].

5.2 Serpent

Serpent was defined by R. Anderson and E. Biham and L. Knudsen in [10] and was the runner-up of the AES block cipher competition. Although it has not been chosen by the NIST during the competition, it was considered to be a close alternative and is still known to be secure from a cryptography point of view.

Serpent was inspired by the idea of bitslice implementations of block ciphers [21] and is optimized to allow a maximum degree of parallelism for software implementations by

exploiting bitslicing techniques. The algorithm represents data as outlined in Section 5.2.1 and exists of two main components: the *Round Transformation* as described in Section 5.2.2 and the *Key Scheduler* explained in Section 5.2.3.

5.2.1 Data Representation

The Serpent cipher is a 32-round substitution-permutation network that operates on four 32-bit words which results in a block size of 128 bits. All values used throughout the operations of the cipher are represented as bitstreams in *little-endian*. So the first bit of a word (bit_0) is the least significant bit and the last bit of a word (bit_{31}) is the most significant one. Similarly, the first word ($word_0$) of a block is the least significant word and the last word ($word_3$) of a block is the most significant one.

5.2.2 The Cipher

The cipher of Serpent is formally defined in two versions: a *non-bitsliced* version and a *bitsliced* version that exploits bitslicing techniques. The structure of both algorithm versions is very similar. Figure 5.6 gives an overview of the bitsliced version of Serpent. It mainly consists of 32 rounds that include a *key-mixing stage*, a *substitution stage*, and an *avalanche stage* (i.e., a stage where a linear transformation takes place). In the last round the linear transformation is omitted and replaced by an additional key-mixing operation. The non-bitsliced version in addition includes an *input permutation (IP)*, and a *final permutation (FP)* that do not have any cryptographic significance. They are only used to transform the input block into a form, that allows to exploit bitslicing methods and to finally transform it back to the conventional form again. The following algorithm description will focus on the bit-sliced version of Serpent. For details on the non-bitsliced version we refer to [10].

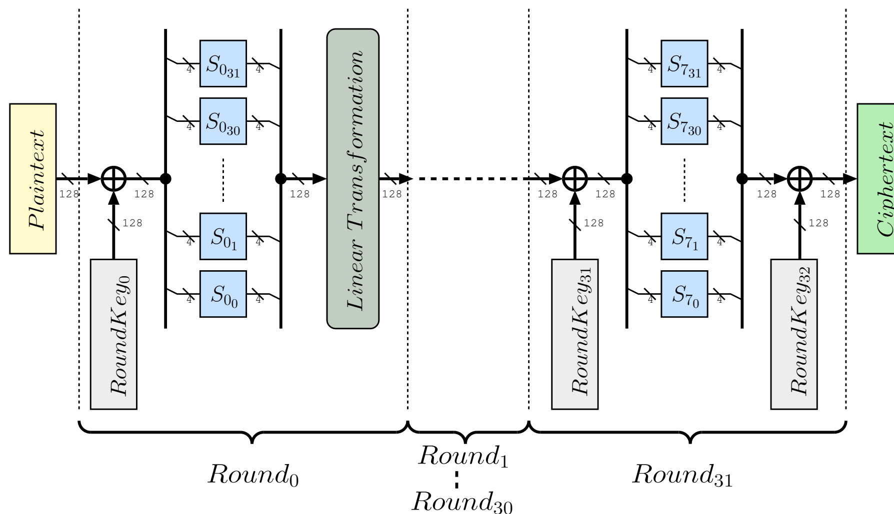


Figure 5.6: Overview of the bitsliced version of Serpent.

The principal structure is defined by the following equation:

$$\begin{aligned} B_0 &:= P \\ B_{i+1} &:= R_i(B_i) \\ C &:= B_{32} \end{aligned} \tag{5.3}$$

where

$$
\begin{aligned}
R_i(X) &:= L(S_j(X \oplus K_i)) && i = 0, \ldots, 30 \\
R_i(X) &:= S_j(X \oplus K_i) \oplus K_{32} && i = 31
\end{aligned}
\tag{5.4}
$$

Initial and Final Permutation

The *initial permutation* and the *final permutation* do not have any cryptographic significance and are only used to transform the input block into a form that can exploit bitslicing methods and back to the conventional form again when using the non-bitsliced version of Serpent. Parameters for the initial and final permutation are given in Appendix B.1.

Key-Mixing Stage

Each round of Serpent starts with a *key-mixing stage*, where a 128-bit round-key K_i is XORed with the input B_i of round R_i.

Substitution Stage

Next follows a *substitution stage*, which uses S-boxes that perform a 4-bit to 4-bit substitution. Overall, Serpent defines eight different S-Boxes. For each round R_i with $i \in \{0, \ldots, 31\}$ a single S-box version S_j, where $j := i \mod 8$, that is replicated 32 times is applied. For example for R_0, the first copy of the S-box S_0 take the bits $0, 1, 2, 3$ of $B_0 \oplus K_0$ and returns the first four bits of an intermediate result. Tables for the S-boxes are given in Appendix B.1.

Linear Transformation

Finally, during the *avalanche stage* a linear transformation is used to maximize the number of bit-changes a single bit-change of the input has. The *linear transformation* is used to linearly mix four 32-bit words by:

$$
\begin{aligned}
X_0, X_1, X_2, X_3 &:= S_i(B_i \oplus K_i) \\
X_0 &:= X_0 \lll 13 \\
X_2 &:= X_2 \lll 3 \\
X_1 &:= X_1 \oplus X_0 \oplus X_2 \\
X_3 &:= X_3 \oplus X_2 \oplus (X_0 \ll 3) \\
X_1 &:= X_1 \lll 1 \\
X_3 &:= X_3 \lll 7 \\
X_0 &:= X_0 \oplus X_1 \oplus X_3 \\
X_2 &:= X_2 \oplus X_3 \oplus (X_1 \ll 7) \\
X_0 &:= X_0 \lll 5 \\
X_2 &:= X_2 \lll 22 \\
B_{i+1} &:= X_0, X_1, X_2, X_3
\end{aligned}
\tag{5.5}
$$

where $\ll$ denotes an arithmetic left shift and $\lll$ denotes a circular left shift.

Note again, that in the last round of the cipher, the linear transformation is omitted and replaced by another key-mixing operation.

Decryption

The Serpent decryption differs from the encryption in that the inverse of all functions have to be used. First, the inverse of the S-Boxes have to be used in reverse order. So for round R_i with $i \in \{0, \dots, 31\}$ the S-box version S_j^{-1}, where $j := (32 - i + 7) \mod 8$ is applied. Furthermore the inverse linear transformation is used and the round keys are applied in reverse order. Consequently, encryption and decryption can share only very little resources.

5.2.3 Key Scheduler

The key schedule is relatively complex and takes a user key K and expands it to 33 128-bit sub-keys denoted by K_i. These sub-keys are required during the round transformations of the cipher. As 33 round keys are required it is necessary to generate 132 32-bit words for the sub-keys. In order to do so, K is padded to 256 bits and then divided into eight 32-bit words denoted as $w_{-8}, \dots, w_{-1}$ and then expanded to an intermediate key (or pre-key) $w_0, \dots, w_{131}$ by applying the following rule:

$$w_j := (w_{j-8} \oplus w_{j-5} \oplus w_{j-3} \oplus w_{j-1} \oplus \phi \oplus j) \lll 11 \tag{5.6}$$

where the constant ϕ is the golden ratio ***0x9e3779b9*** in hexadecimal. The round keys can then be computed, by making use of the S-boxes in the following manner:

$$\begin{aligned}
\{k_0, k_1, k_2, k_3\} &:= S_3(w_0, w_1, w_2, w_3) \\
\{k_4, k_5, k_6, k_7\} &:= S_2(w_4, w_5, w_6, w_7) \\
\{k_8, k_9, k_{10}, k_{11}\} &:= S_1(w_8, w_9, w_{10}, w_{11}) \\
\{k_{12}, k_{13}, k_{14}, k_{15}\} &:= S_0(w_{12}, w_{13}, w_{14}, w_{15}) \\
\{k_{16}, k_{17}, k_{18}, k_{19}\} &:= S_7(w_{16}, w_{17}, w_{18}, w_{19}) \\
&\dots \\
\{k_{124}, k_{125}, k_{126}, k_{127}\} &:= S_4(w_{124}, w_{125}, w_{126}, w_{127}) \\
\{k_{128}, k_{129}, k_{130}, k_{131}\} &:= S_3(w_{128}, w_{129}, w_{130}, w_{131})
\end{aligned} \tag{5.7}$$

The 32-bit values k_j are then concatenated to the 128-bit sub-keys K_i where $i \in \{0, \dots, 32\}$:

$$K_i := \{k_{4i} \,\|\, k_{4i+1} \,\|\, k_{4i+2} \,\|\, k_{4i+3}\} \tag{5.8}$$

Figure 5.7 describes the calculation of the pre-keys required for on round key. Please note, that the calculation of the pre-keys needed for on round key requires eight 32-bit values from previous rounds to be available.

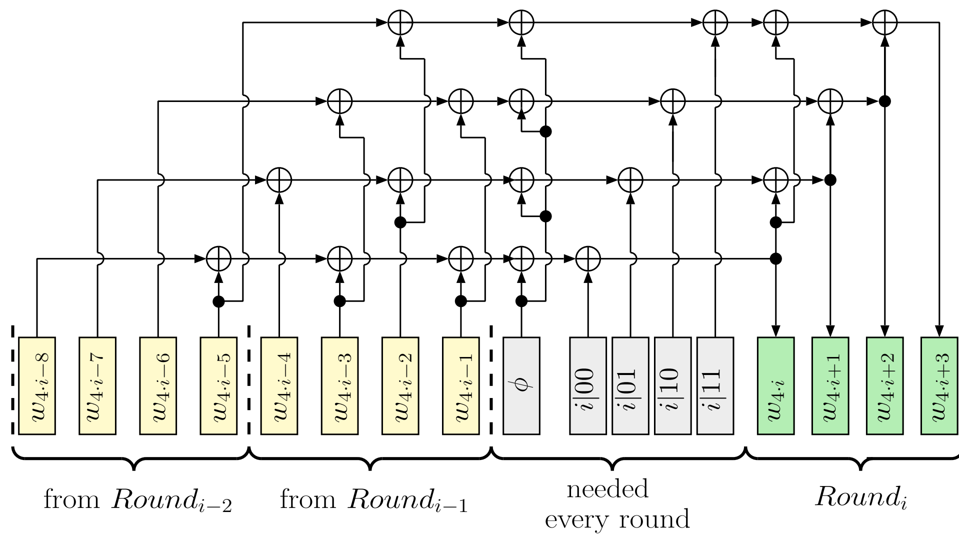


Figure 5.7: Overview of the pre-key calculation of Serpent.

Chapter 6

Authenticated Encryption

An *authenticated encryption (AE)* scheme is a cryptographic primitive based on a shared-key transform which aims at providing *privacy* and *authenticity* of the encapsulated data. A refined version of AE which allows parts of the data to be authenticated but not encrypted is called *authenticated-encryption with associated-data (AEAD)* and was formally defined in [92]. In fact, AEAD is of special interest in networking applications as there often is a need to have a header to appear in plain text, but there is a wish to have this header authenticated. However, the payload needs to be private and authenticated.

Numerous algorithms exist and some, such as the CBC-MAC mode (CCM) [77] and the Galois Counter Mode of operation (GCM) [78] have also been standardized by NIST. However, these are widely seen as unsatisfactory standards that come with some disadvantages that can compromise security [83]. Thus, in 2014, a new competition called *CAESAR* (Competition for Authenticated Encryption: Security, Applicability, and Robustness) was launched by the international cryptologic research community [27]. Its main goal is to identify authenticated ciphers that offer advantages over the Galois Counter Mode of operation (GCM) and that are suitable for widespread adoption. The whole competition is based on a public evaluation. For the first round of the competition, launched in March 2014, 57 authenticated cipher candidates were submitted some of which have already been withdrawn because of security flaws that could be identified. However, as the competition is in a very early stage it is not yet foreseeable which submissions will finally be approved and currently only few hardware implementations of proposals have been published.

Different approaches to authenticate and encrypt a message are discussed in Section 6.1. Section 6.2 gives an overview of properties and features that characterize AE algorithms. Thereafter, follows the description of some selected algorithms for authenticated encryption. Section 6.3 treats the Counter with CBC-MAC mode (CCM), Section 6.4 outlines the Carter–Wegman Counter (CWC) mode [59], Section 6.5 explains the Galois Counter Mode of operation (GCM) and Section 6.6 deals with the Offset CodeBook mode (OCB).

6.1 Approaches to Achieve Authenticated Encryption

There are two main principles to authenticate and encrypt a message. The first way of achieving AE is to use a *generic composition scheme* which works by combining a secure encryption scheme with a keyed-hash (i.e. MAC) scheme. For example the plaintext could be encrypted with AES in Cipher Block Chaining (CBC) mode [76] and then HMAC [75]

could be applied on the ciphertext to compute an authentication tag. The second approach is to use specially designed *integrated authenticated-encryption algorithms* which typically are more efficient then generic composition modes.

In theory there are three different combinations of the *generic composition schemes* for AE.

- **MAC-then-Encrypt:** First MAC the message M with key K_1 to give a tag $\tau = MAC_{K_1}(M)$ and then encrypt the pair (M, τ) with key K_2 to give the cipher text $C = E_{K_2}(M, \tau)$. The final message then is the cipher text (C).

- **Encrypt-then-MAC:** First encrypt the message M with key K_1 to give the cipher text $C = E_{K_1}(M)$ and then apply the MAC with key K_2 such that the tag $\tau = MAC_{K_2}(C)$. The final message then is the pair (C, τ).

- **Encrypt-and-MAC:** First encrypt the message M with key K_1 to give $C = E_{K_1}(M)$ and additionally MAC M with key K_2 to give the tag $\tau = MAC_{K_2}(M)$. The final message then is the pair (C, τ).

However, in [13] Bellare showed that the *Encrypt-then-MAC* scheme used with a provable-secure encryption scheme and a provable-secure MAC scheme provides the best security for AE among the three proposed generic combinations. Still, also the MAC-then-Encrypt and Encrypt-and-MAC schemes can be secure but they often rely on special details of the underlying encryption and MAC schemes. *Thus, among the generic composition modes the Encrypt-then-MAC scheme is the preferred choice.*

There are three main approaches to realize an *integrated authenticated-encryption algorithm*. The first method is to use a block cipher in a special mode of operation for authenticated encryption. NIST has standardized two block cipher modes of operation for AE, namely, CBC-MAC mode (CCM) [77] and the Galois Counter Mode of operation (GCM) [78]. Another, widely known AE mode is the Offset CodeBook (OCB) mode [63]. The second method is to use a stream cipher and to divide the keystream into two parts: one for the encryption and one for the authentication of the message. Grain-128a [4] is a typical example for this approach. The third strategy is to develop a dedicated authenticated encryption algorithm. Examples for this approach are Helix [42], Hummingbird-2 [40] and AEGIS [110].

6.2 Properties and Features of Authenticated Encryption Algorithms

Depending on the designated operational scenarios and chosen design rationales authenticated-encryption algorithms have different characteristics and features. This section briefly discusses the most important ones with having high-speed hardware implementations and networking applications in mind. Using these characteristics can help to identify an appropriate algorithm for a desired application.

Type of Scheme. On the top level authenticated-encryption algorithms can be categorized according to the different approaches to achieve AE as outlined in Section 6.1: *generic*

composition, *block-cipher mode of operation*, *stream-cipher based* or *dedicated authenticated-encryption algorithms*. In addition, based on the required number of passes over the message M, the algorithms are classified as so-called *single-pass* schemes or *two-pass* schemes. A single-pass scheme only requires one pass over the message while a two-pass scheme requires two passes respectively. Other important properties are the supported lengths for message blocks, keys and authentication-tags and the maximum length of the input message.

Security. Many AE algorithms are backed with security proofs that (theoretically) guarantee a specific level of security under specific assumptions and restrictions for an adversary. Apart from this information-theoretical security attributes some algorithms also include measures for actual implementations that hamper side channel attacks. In addition, the algorithms can come with a certain level of misuse resistance as pointed out later in this section.

Parallelizable. *Parallelization* is one of the most important concepts to reach high throughput. It is a form of replication by providing q instances of the same functional unit for f and letting them process independent inputs in parallel. Here, the throughput as well as the hardware consumption are roughly multiplied by q. Note, that it is strictly necessary that the inputs to the functional units are independent from each other to allow parallelization. For high-speed applications it is necessary, that encryption and decryption can be parallelized.

Online. An AE algorithm is said to provide *online* encryption and decryption, if it is able to output blocks while it is receiving blocks and without prior knowledge of the overall message length. More precisely, the ith Output block O_i should only depend on the first i input blocks $I_1, \ldots, I_i$ and the secret key K. This is a desirable feature, especially in networking and resource-constraint applications as it reduces memory requirements.

Inverse Free. An authenticated-encryption algorithm is said to be *inverse free* if encryption and decryption rely on the same functions f and no inverses f^{-1} are required. This is of special interest for hardware implementations that have to deal with limited resources as the same data path can be used for encryption and decryption.

AEAD. Authenticated-encryption algorithms allowing *AEAD* is of special interest in networking applications as there often is a need to have a header to appear in plain text, but there is a wish to have this header authenticated while the payload needs to be encrypted and authenticated.

Misuse Resistance. Many AE schemes rely on the use of a user-applied state called nonce to prevent leaking any information on the plaintext except its length. Often security proofs assume that these nonces are correctly generated and used and that there is no *nonce-misuse*. The security of many AE algorithms rely on the assumption that the nonce is not reused for the same key, and do not provide authenticity and privacy in case of misuse [44]. The past has shown, that the use of nonces is frequently implemented faulty [22, 58]. Thus, it is desirable, that an AE algorithm provides a decent level of security even if nonces are repeated.

Standard AE security definitions assume that there is no *decryption misuse* and that an adversary does not get any information on the output of decrypted ciphertexts that fail in authenticity verification. However, in practice this is not always easy to ensure. Thus, it is desirable, that an AE algorithm provides a decent level of security even if decryptions of ciphertexts that do not pass the authenticity check are leaking.

Message Overhead. The message overhead is determined by length of the authentication tag T and the extension of the ciphertext. Ideally, the length of the plaintext P is equal to the length of the resulting ciphertext C.

Additional Features. AE algorithms can come with a variety of additional features. They can for example support variable authentication tag length, incremental MACs[1] or facilitate the implementation of counter measures against side channel attacks. In addition they can minimize the cost of key changes or nonce changes.

Apart from the characteristics and features addressed in this section many more exist that are relevant for special use cases like software implementations or resource-constraint implementations. Section 10.3 in detail analyzes some authenticated-encryption schemes based on their characteristics and features towards their feasibility to reach a throughput of 100 Gbit/s.

6.3 CCM

The Counter with CBC-MAC mode (CCM) [53, 77] was designed as non-patented alternative to OCB and is part of the IEEE 802.11i standard. It uses the "MAC-then-Encrypt" approach. As outlined in Figure 6.1 the CCM mode combines the CBC-MAC scheme for data authentication and a variation of the Counter (CTR) mode for data encryption.

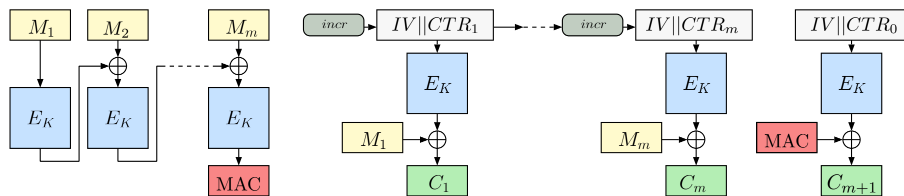


Figure 6.1: Overview of CCM. Here *incr* denotes an increment by one modulo 2^{32}.

First, the MAC is computed over the entire message using the CBC-MAC. This step is not parallelizable. Second, the message and the MAC is encrypted using a CTR mode that takes IV and a counter value as input. For decryption, first the message and the MAC is decrypted and then the MAC is computed over the resulting plaintext and compared with the transmitted MAC.

CCM is non-patented, non-parallelizable, provable secure mode with AEAD feature. It has a comparably low performance in hardware and has received some criticism concerning efficiency, complexity and security claims [91]. So, overall, as *CCM* is non-parallelizable it is *not suited for the throughput* required in the QCrypt project.

[1] An incremental MAC allows to recompute the authentication tag at a cost proportional to the amount of changed blocks of a message.

6.4 CWC

The Carter–Wegman Counter (CWC) mode [59] created by Kohno, Viega, and Whiting combines the CTR mode for data encryption with a Carter-Wegman universal hashing function over $GF(2^{127} - 1)$ for a data authentication. It basically uses the "Encrypt-then-MAC" approach with both steps chosen to be parallelizable. Please refer to Figure 6.2 for an overview of the CWC encryption.

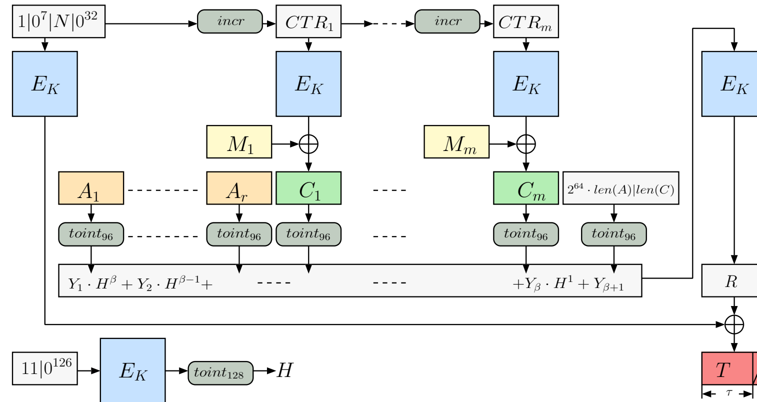


Figure 6.2: Overview of CWC. Here *incr* denotes an increment by one modulo 2^{32} and $toint_{96}$ denotes a reduction to 96 bits.

First, the message is encrypted using CTR mode with a nonce IV as the initial counter value which gives the ciphertext C. Next, the authentication part starts by calculating the CWC-Hash value for associated data A and C. Then, the intermediate authentication tag R is calculated by encrypting this hash value. The final authentication tag T is then generated by XORing R with the padded and encrypted IV.

CWC is non-patented, parallelizable, provable secure. Due to its comparably low efficiency it has widely been overcome by GCM mode that will be explained in the next section.

6.5 Galois Counter Mode of Operation

The Galois Counter Mode of operation (GCM) was invented by David A. McGrew and John Viega and is standardized by NIST [78]. GCM is a block-cipher mode of operation defined for block ciphers with 64-bit and 128-bit block size. It is based on the CTR mode and a Carter-Wegman general hash function over $GF(2^{128})$ called *GHASH* and allows AEAD. As it achieves AE by two passes over the message M it is a so-called *two-pass* scheme.

6.5.1 Authenticated Encryption

The GCM authenticated encryption algorithm as shown in Figure 6.3 expects a secret key K with appropriate length for the underlying block cipher and an initial vector IV of arbitrary length between 1 and 2^{64} bits, where a length of 96 bits is recommended for efficient implementations, as inputs. Note, that the IVs need to be distinct for a fixed key K. Furthermore, the algorithm requires a plaintext P with a length between 0 and $2^{39} - 256$ bits and some additional authenticated data (AAD) denoted as A. The AAD

is data that should be authenticated but not encrypted and is allowed to be between 0 and 2^{64} bits of length. For an underlying block cipher with 128-bit block size P is split into blocks of 128 bits such that $P = P_1, P_2, \ldots, P_{n-1}, P_n^*$ where the last block P_n^* does not need to be a full block whose length is denoted by u. Equally, the AAD is split into blocks such that $A = A_1, A_2, \ldots, A_{m-1}, A_m^*$ where the length of the last block is denoted by v.

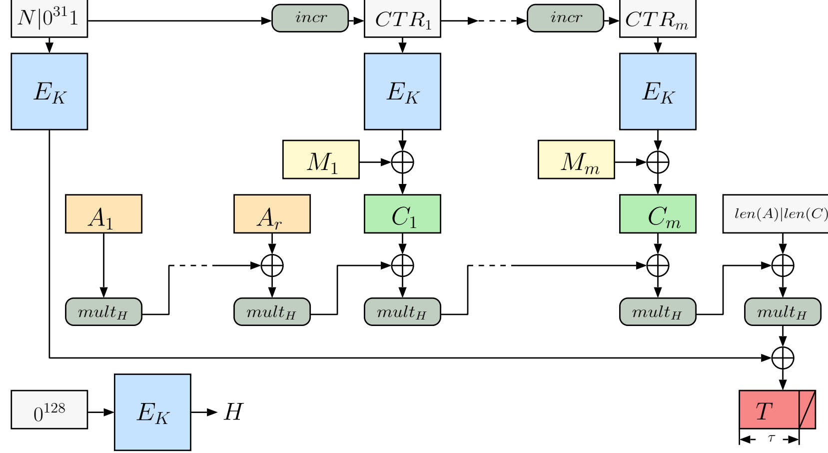


Figure 6.3: Overview of the GCM authenticated encryption operation. Here *incr* denotes an increment by one modulo 2^{32} and $mult_H$ denotes a multiplication with H in $GF(2^{128})$.

Formally, the authenticated encryption is defined by the following equations:

$$
\begin{aligned}
H &= E(K, 0^{128}) \\
Y_0 &= \begin{cases} IV \| 0^{31}1 & \text{if } \mathrm{len}(IV) = 96 \\ \mathrm{GHASH}(H, \{\}, IV) & \text{else.} \end{cases} \\
Y_i &= \mathrm{incr}(Y_{i-1}) \text{ for } i = 1, \ldots, n \\
C_i &= P_i \oplus E(K, Y_i) \text{ for } i = 1, \ldots, n-1 \\
C_n^* &= P_n^* \oplus \mathrm{MSB}_u(E(K, Y_n)) \\
T &= \mathrm{MSB}_\tau(\mathrm{GHASH}(H, A, C) \oplus E(K, Y_0))
\end{aligned}
\tag{6.1}
$$

where $\oplus$ denotes a bitwise-exclusive-or (XOR) operation, $\|$ denotes a bitwise-concatenation, the expression 0^l denotes a string of l zero bits and the expression $\{\}$ represents a bit string of zero length. The function $len()$ returns the number of bits in its argument, the function $MSB_\tau(S)$ returns the first τ most significant bits of the bit string S and the $incr()$ function increments its argument by one modulo 2^{32}. The function $GHASH()$ is a polynomial universal hash and is described more detailed in Section 6.5.3. As output the GCM authenticated encryption algorithm delivers a ciphertext C where $\mathrm{len}(C) = \mathrm{len}(P)$ which is split into 128-bit blocks such that $C = C_1, C_2, \ldots, C_{n-1}, C_n^*$, where the length of the last block is denoted by u, and an authentication tag T with a length τ where $0 \leq \tau \leq 2^{64}$.

6.5.2 Authenticated Decryption

The structure of the authenticated decryption as illustrated in Figure 6.4 is very similar to the structure of the authenticated encryption, with the only difference, that the ciphertext

can directly be applied to the hash step and the decryption step.

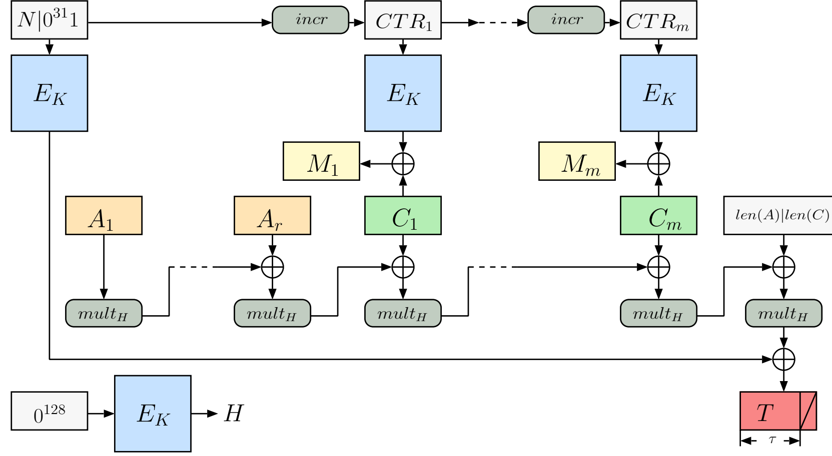


Figure 6.4: Overview of GCM authenticated decryption operation. Here *incr* denotes an increment by one modulo 2^{32} and $mult_H$ denotes a multiplication with H in $GF(2^{128})$.

The authenticated decryption operation, requires five inputs: the secret key K, an initial vector IV, a ciphertext C, some additional authenticated data A and an authentication tag T and is defined as follows:

$$
\begin{aligned}
H &= E(K, 0^{128}) \\
Y_0 &= \begin{cases} IV \| 0^{31}1 & \text{if } \operatorname{len}(IV) = 96 \\ \operatorname{GHASH}(H, \{\}, IV) & \text{else.} \end{cases} \\
T' &= \operatorname{MSB}_\tau(\operatorname{GHASH}(H, A, C) \oplus E(K, Y_0)) \\
Y_i &= \operatorname{incr}(Y_{i-1}) \text{ for } i = 1, \ldots, n \\
P_i &= C_i \oplus E(K, Y_i) \text{ for } i = 1, \ldots, n \\
P_n^* &= C_n^* \oplus \operatorname{MSB}_u(E(K, Y_n))
\end{aligned} \tag{6.2}
$$

The tag T' which is computed during the decryption process is compared to the tag T coming with the ciphertext C. If both tags exactly match the authenticated decryption operation returns the plaintext. Otherwise it returns a the special symbol ***FALSE***.

6.5.3 GHASH

The GHASH function $\operatorname{GHASH}(H, A, C) = X_{m+n+1}$ belongs to the class of Carter-Wegman polynomial universal hashes and is defined by Equation 6.3 where C is the ciphertext, A is the data to be authenticated and u and v represent the missing bits to a full block of 128 bits.

$$
X_i = \begin{cases}
0 & \text{for } i = 0 \\
(X_{i-1} \oplus A_i) \cdot H & \text{for } i = 1, \ldots, m-1 \\
(X_{m-1} \oplus (A_m^* \| 0^{128-v})) \cdot H & \text{for } i = m \\
(X_{m+n-1} \oplus (C_m^* \| 0^{128-u})) \cdot H & \text{for } i = m+n \\
(X_{m+n} \oplus (\operatorname{len}(A) \| \operatorname{len}(C))) \cdot H & \text{for } i = m+n+1
\end{cases} \tag{6.3}
$$

The multiplication of two elements X and Y in Equation 6.3 denoted as $X \cdot Y$ are multiplications in $GF(2^{128})$. Please refer to [70] and [78] for details on the multiplication in $GF(2^{128})$ used in GCM.

6.6 Offset CodeBook (OCB)

The offset codebook (OCB) block cipher mode of operation which has been published by Rogaway et al. [93] in 2001 is a combined AE scheme that is able to achieve AE by just a single pass over the message M. Thus OCB is a so-called *single-pass* scheme. It is strongly related to the Integrity Aware Parallelizable Mode (IAPM) by C. Jutla [55] and already three different versions have been made public since 2003. Throughout the remainder of this paper, when speaking of OCB, we solely refer to the third version of it, i.e., OCB3 [63] that is also defined in RFC 7253 [61] which an informational RFC. In addition OCB was submitted to the CAESAR challenge [62]. OCB is patented but its patents are freely licensed over a large space: open-source software, non-military software, and OpenSSL [62].

6.6.1 Authenticated Encryption

To start the authenticated encryption scheme according to OCB, a plaintext message, denoted by M, gets split into m different blocks, each of length[2] n and an optional block M_* of length smaller than n as follows:

$$M = \begin{cases} M_1, \ldots, M_m, & \text{if } |M| = k \cdot n \text{ and } k \in \mathbb{N}, \\ M_1, \ldots, M_m, M_*, & \text{else.} \end{cases}$$

In addition it accepts additional authenticated data, denoted by A, that gets split into p different blocks, each of length n and an optional block A_* of length smaller than n as follows:

$$A = \begin{cases} A_1, \ldots, A_p, & \text{if } |A| = k \cdot n \text{ and } k \in \mathbb{N}, \\ A_1, \ldots, A_p, A_*, & \text{else.} \end{cases}$$

As outlined in Figure 6.5, OCB can be divided into two main parts: an authentication part and an authenticated-encryption part.

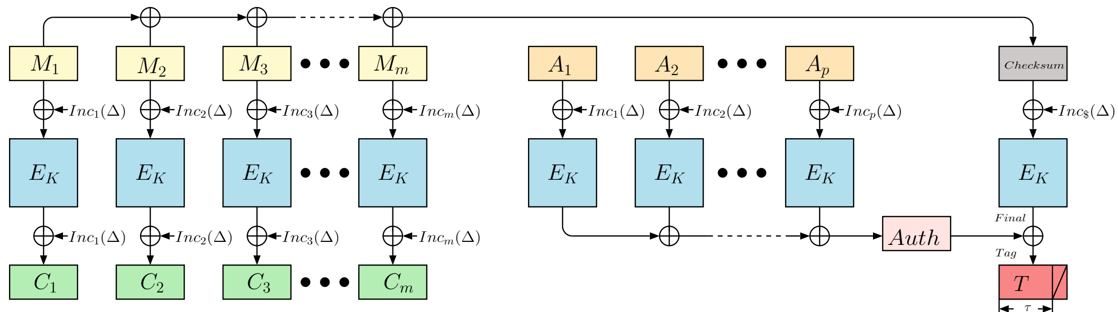


Figure 6.5: Overview of the authenticated encryption of OCB assuming full final blocks. Here $inc_i(\Delta)$ denotes the increment of the Δ-values according to line 3 of Algorithm 6.6.4 and line 5 of Algorithm 6.6.1 respectively.

[2] We refer to the length of x in bits using the following notation: $|x|$

The authentication part processes the message blocks A_i of the input message, that only need to be authenticated and compresses them to an intermediate hash value $Auth$. The authenticated-encryption part is used for message blocks M_i that need to be authenticated and encrypted and encrypts these to the ciphertexts C_i. Finally, the resulting authentication tag T is computed by encrypting the *checksum* over all message blocks $M_0, \ldots, M_m$ is encrypted, XORed with the intermediate hash value $Auth$ and finally truncated to the desired bit length τ.

Algorithm 6.6.1 gives an exact description of the authenticated encryption according to OCB. The characters $\|$, $\oplus$, and $\wedge$ denote the concatenation-, bitwise-exclusive-or-, and bitwise-and-operation, respectively. Furthermore, *ntz(i)* describes the number of trailing zeroes of i in binary representation. We use $\varnothing$ to represent both an empty binary string of length n (cf. line 3), and an *empty set* as within line 11. The listings for the procedures *Setup*, *Init*, and $Hash_K$ used throughout Algorithm 6.6.1 are given in Algorithm 6.6.2, Algorithm 6.6.3 and in Algorithm 6.6.4 and are described in the following.

Algorithm 6.6.1 OCB authenticated encryption.

Input: Message M of m blocks length, Message block length n, Cipher key K, Nonce N, Associated data A of p blocks length, Tag length τ

Output: Ciphertext C, Authentication tag T

1: **if** $|N| \geq n$ **then return** INVALID
2: $\{M_1, \ldots, M_m, M_*\} \leftarrow M$, with $|M_i| = n$ and $|M_*| < n$
3: $Checksum \leftarrow \varnothing; C \leftarrow \varnothing$
4: $L_*, L_\$, L[0] \ldots L[\lfloor log_2(m) \rfloor] \leftarrow Setup(K, m)$
5: $\Delta \leftarrow Init(N, n, K)$
6: **for** $i = 1$ to m **do**
7: $\quad \Delta \leftarrow \Delta \oplus L[ntz(i)]$
8: $\quad C \leftarrow C \,\|\, E_K(M_i \oplus \Delta) \oplus \Delta$
9: $\quad Checksum \leftarrow Checksum \oplus M_i$
10: **end for**
11: **if** $M_* \neq \varnothing$ **then**
12: $\quad \Delta \leftarrow \Delta \oplus L_*$
13: $\quad Pad \leftarrow E_K(\Delta)$
14: $\quad C \leftarrow C \,\|\, M_* \oplus (Pad \wedge (2^{|M_*|} - 1))$
15: $\quad Checksum \leftarrow Checksum \oplus M_*10^*$, with
$\qquad M_*10^* = M_* \| 1 \| 0 \ldots 0$, such that $|M_*10^*| = n$
16: **end if**
17: $\Delta \leftarrow \Delta \oplus L_\$$
18: $Final \leftarrow E_K(Checksum \oplus \Delta)$
19: $Auth \leftarrow Hash_K(A)$
20: $Tag \leftarrow Final \oplus Auth$
21: $T \leftarrow Tag \wedge (2^\tau - 1)$
22: **return** $C \,\|\, T$

The OCB algorithm starts with a setup and initialization step (cf. line 4 and 5). The setup step as given in algorithm 6.6.2 and outlined in Figure 6.6 includes the computation of the $L[..]$-values.

Algorithm 6.6.2 Table value calculations.

Input: Cipher key K, Number of message blocks m
Output: $Setup(K, m)$
1: $L_* \leftarrow E_K(\varnothing)$
2: $L_\$ \leftarrow double(L_*)$
3: $L[0] \leftarrow double(L_\$)$
4: **for** $i = 0$ to $\lfloor log_2(m) \rfloor$ **do**
5: $\quad L[i] \leftarrow double(L[i-1])$
6: **end for**
7: **return** $L_*, L_\$, L[0] \ldots L[\lfloor log_2(m) \rfloor]$

Here the *double*-procedure is defined according to:

$$double(X) = (X \ll 1) \oplus (msb(X) \cdot 0x87) \tag{6.4}$$

where $msb(X)$ represents the most significant bit of X using binary representation.

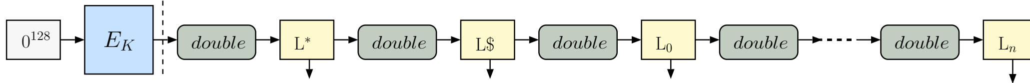


Figure 6.6: OCB calculation of the $L[..]$-values.

During the initialization step the initial offset Δ is determined according to Algorithm 6.6.3. Here, $x \ll i$ denotes a left shift operation of x by i bits.

Algorithm 6.6.3 Initial offset (Δ) calculation.

Input: Nonce N, Message block length n, Cipher key K
Output: $Init(N, n, K)$
1: $Nonce \leftarrow 1 \,\|\, N$
2: $Top \leftarrow (1^{122} \,\|\, 0^6) \wedge Nonce$
3: $Bottom \leftarrow Nonce \wedge (2^6 - 1)$
4: $Ktop \leftarrow E_K(Top)$
5: $Stretch \leftarrow Ktop \,\|\, (Ktop \oplus (Ktop \ll 8))$
6: $\Delta \leftarrow (Stretch \ll Bottom) \wedge (2^n - 1)$
7: **return** Δ

Figure 6.7 outlines the initialization step that calculates the initial Δ-value.

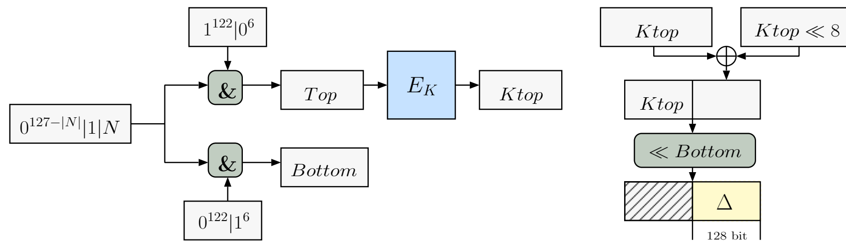


Figure 6.7: Overview the calculation of the initial Δ value during the initialization step.

Thereafter, each message block can be processed independently of each other (line 6 to 16) which allows the parallelization of OCB. Algorithm 6.6.4 describes the calculation of $Hash_K(A)$ which gives the intermediate hash *Auth*. Since the *Setup* procedure already gets called during the actual encryption process of OCB (cf. Algorithm 6.6.1, line 4), line 3 in Algorithm 6.6.4 can be omitted as long as the table values $L[..]$ are globally available. Note, that the *Hash*-procedure uses a Δ-value different from the Algorithm 6.6.1.

Algorithm 6.6.4 Authentication hash ($Hash_K(A)$) calculation.

Input: Associated data A, Associatd data block length q, Cipher key K
Output: $Hash_K(A)$

1: $\{A_1, \dots, A_p, A_*\} \leftarrow A$, with $|A_i| = q$ and $|A_*| < q$
2: $Sum \leftarrow \varnothing$; $\Delta \leftarrow \varnothing$
3: $L_*, L[0] \dots L[\lfloor log_2(p) \rfloor] \leftarrow Setup(K, p)$
4: **for** $i = 1$ to p **do**
5: $\quad \Delta \leftarrow \Delta \oplus L[ntz(i)]$
6: $\quad Sum \leftarrow Sum \oplus E_K(A_i \oplus \Delta)$
7: **end for**
8: **if** $A_* \neq \varnothing$ **then**
9: $\quad \Delta \leftarrow \Delta \oplus L_*$
10: $\quad Sum \leftarrow Sum \oplus E_K(A_*10^* \oplus \Delta)$, with
$\qquad A_*10^* = A_*\|1\|0 \dots 0$, such that $|A_*10^*| = q$
11: **end if**
12: **return** Sum

Finally, the authentication tag T is determined throughout line 17 to 21. Here τ represents the bit length of the authentication tag T.

When using a counter for the nonce N, the calculation of the initial offset Δ requires a block cipher call only every $64th$ initialization. This is due to the fact that the least significant six bits of N are set to zero before passing it to the block cipher (cf. line 2 of Algorithm 6.6.3). This fact together with the parallelizable processing of the message blocks, makes OCB suitable for high-throughput applications.

6.6.2 Authenticated Decryption

The OCB decryption as shown in Figure 6.8, requires encryption and decryption functionality of the underlying block cipher. During the initialization and setup phase, as well as for the calculation of the intermediate hash value *Auth* and the computation of the final authentication tag T, similar to the OCB authenticated-encryption, cipher encryptions are needed. However, the encryptions in line 8 of Algorithm 6.6.1 have to be replaced with cipher decryptions. Appendix B.2 includes the algorithm description for the OCB authenticated decryption.

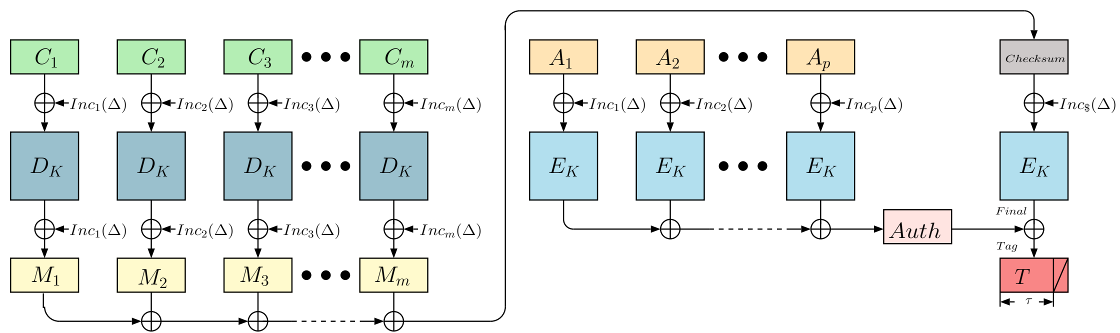


Figure 6.8: Overview of the authenticated decryption of OCB. Here $inc_i(\Delta)$ denotes the increment of the Δ-values according to Line 3 of Algorithm 6.6.4 and Line 5 of Algorithm 6.6.1 respectively.

Chapter 7

Reconfigurable Hardware

This chapter gives a short introduction on reconfigurable hardware and field-programmable logic (FPL) devices. Section 7.1 introduces the idea of reconfigurable hardware and introduces to the different types of FPL devices. A prominent example of FPL device are field-programmable gate arrays (FPGAs), which this work is based on. Therefore Section 7.2 gives an overview of the architecture of FPGAs and its sub-components and explains how these devices can be programmed and configured.

7.1 Introduction to Reconfigurable Hardware

In applications that have specific requirements concerning issues like performance or energy consumption, general-purpose integrated circuits like microcontrollers or digital signal processors (DSPs) are often unsuitable and dedicated integrated circuits (ICs) have to be designed. Application-specific integrated circuits (ASICs) can provide one possible solution in this case. However, they have a long time-to-market and due to the high nonrecurring engineering costs ASICs require high volumes to be economic. Furthermore, once the ASIC is produced the design is fixed and changes to the hardware are not possible. In contrast, FPL devices which could be viewed as "soft hardware" allow to be reconfigured and offer fast turnaround times and thus provide an attractive alternative in cases where smaller volumes are expected, for prototyping, when a short time-to-market is required and when frequent modifications are likely [56].

Reconfigurable hardware emerged from programmable read-only memory (PROM) and programmable logic devices (PLDs) and were initially used for glue logic. Today's FPL devices are extremely powerful and can mainly be divided into *complex programmable logic devices (CPLDs)* and *field programmable gate arrays (FPGAs)*. The main difference between CPLDs and FPGAs is of architectural nature. CPLDs combine hundreds of identical subcircuits, each containing a simple layer of programmable logic and layer of flipflops and local feedback paths, with a programmable interconnect matrix network. However, for a given task, this form of organization requires artificial partitioning into a bunch of cooperating subcircuits which does not encourage efficiency. Typically, CPLDs provide less gates than FPGAs but keep their configuration on power down and ease the prediction of path delays. As FPGAs usually do not keep their configuration on power down, CPLDs are often used to load configuration data for FPGAs on startup. For a more detailed introduction to the history and the details of reconfigurable hardware please refer

to [67, 112].

As this work was realized using an FPGA the next section will give an introduction to FPGAs and explain its principle and structure.

7.2 Field-Programmable Gate Arrays (FPGAs)

Although many different FPGA designs and architectures by different vendors exist, they basically all follow the architecture illustrated in Figure 7.1. The architecture typically consists of an array of logic blocks, called logic array blocks[1], configurable[2] input/output (I/O) cells and a configurable interconnect matrix that is used to route the signals between the logic blocks and I/O blocks. Furthermore, it contains a clock circuitry that is used to drive the clocks signals for the flipflops included in the logic blocks. In addition, many modern FPGAs include special purpose ressources such as DSPs, adders, multipliers, decoders, memory and even embedded general purpose processors. Note, that the regular structure of FPGAs is very similar to gate array ASICs.

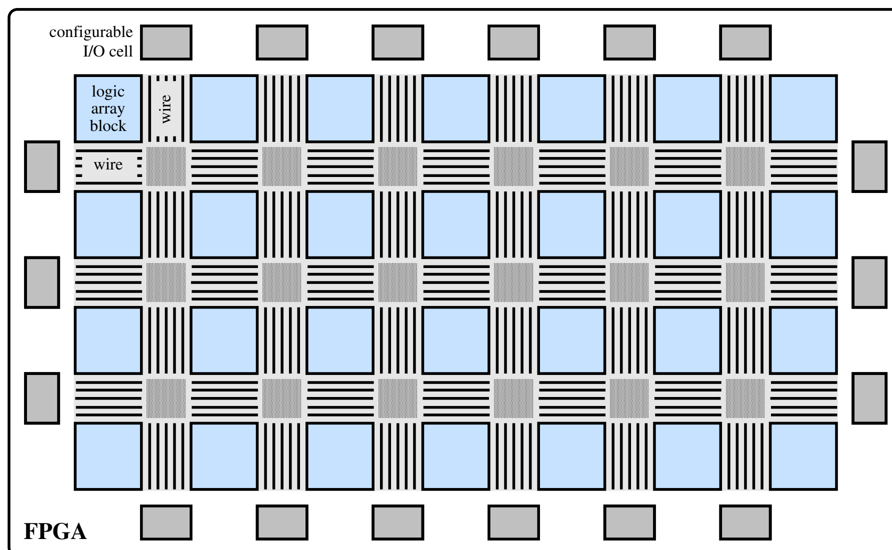


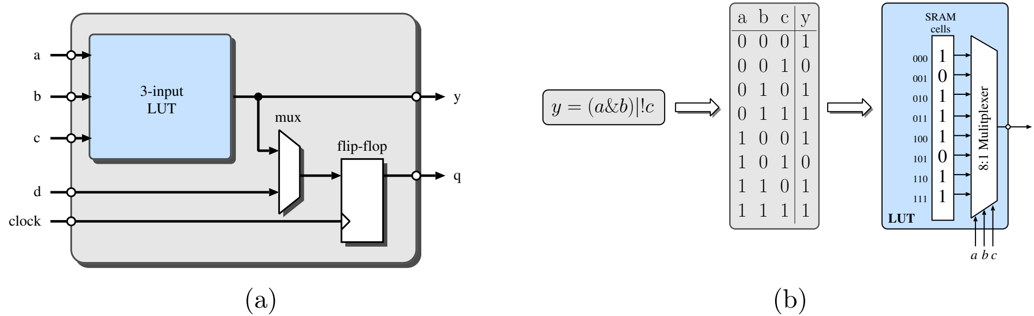
Figure 7.1: General architecture of an FPGA.

7.2.1 Logic Array Blocks (LABs) and Arithmetic Logic Modules (ALMs)

The *Logic Array Blocks (LABs)* contain an array of smaller logic blocks. Note, that the naming and exact architecture of the logic blocks varies from vendor to vendor. Altera calls the logic blocks Arithmetic Logic Module (ALM) while Xilinx calls them Slices. However, as this work is based on an Altera FPGA, we will stick with the name ALM in the following. Usually ALMs contain two or four smaller structures that are similar and share some signal. Typically, these sub-structures feature a n-input *look-up table (LUT)* or n-LUT, which is a memory (typically SRAM) that can be used to produce arbitrary n-input Boolean equations. In addition, they contain a register that can be configured to act as a flipflop or latch

[1] Depending on the FPGA vendor the naming and architecture of the array of logic blocks may vary. Altera calls them Logic Array Blocks (LABs) and Xilinx names them Configurable Logic Blocks (CLBs).

[2] The term “programmable” is often used synonymous to the term “configurable”.



(a) (b)

Figure 7.2: Overview of a simplified configurable logic block (a) and the configuration of a LUT (b).

and multiplexers that are used to route the logic within the ALM and to route incoming and outgoing signals. Figure 7.2a shows a simplified form of a typical sub-structure of an ALM which consists of a 3-LUT, that can be built out of eight SRAM cells as illustrated in Figure 7.2b, a register and a multiplexer that can be used to select the register's input. Note, that the output of the LUT does not need to go through the register which allows to combine multiple ALMs to create complex Boolean functions. This is an advantage over CPLDs.

ALMs on modern FPGAs, such as the Altera Stratix IV and Altera Stratix V devices or the Xilinx Virtex 7 device family, are much more sophisticated and may contain up to 8-input look-up tables and adders and offer many configuration possibilities. For example, the n-input LUTs can also be configured to be used as $n \times 1$-bit RAM or n-bit shift registers. The FPGA vendors provide detailed information on their FPL devices in the datasheets and a more general and concentrated overview is given in [67, 94, 112]. Note, that due to different sets of features and configurations for ALMs of different vendors and even for different device families of the same vendor, it is hard to compare their performance and resource consumption for given algorithms.

7.2.2 Configurable I/O Blocks

The *configurable I/O blocks* allow to bring signals onto the device and also to bring them back off again. They can be configured to be three-state or open collector and also allow to control the slew rate. Additionally, they can be programmed to work with different voltage levels in order to be capable to interact with CMOS or TTL devices. Furthermore, as routing delays are significant in FPGAs, they typically contain a register to be able to either meet hold times routing an incoming signal through the internal interconnect to a flip-flop inside one of the devices ALMs or to reduce the delay for outgoing signals to allow moderate setup times for interfacing devices. This allows the FPGA to run at maximum Speed.

7.2.3 Programmable Interconnect and Clock Circuitry

The *programmable interconnect* consists of multiple hierarchies. First, each ALM is directly connected to its nearest neighbors which allows to realize logic that is too complex for single ALM without introducing a lot of routing delay. Second, another level of routing ressources bypasses a number of ALMs before leading into switch matrices which provide the routing of signals. However, these switches introduce a noticeable delay. The third

level of interconnect is long lines which can be used to connect ALMs that are distant to each other but have are critical in matters of delay. Still, FPGAs have a high routing delay which often can be greater than the logic delay.

The *clock circuitry* consists of special I/O cells with special clock buffers that are used to drive the internal clock lines that are optimized for low skew times and fast propagation.

7.2.4 Programming and Configuration of FPGAs

FPGAs are programmed using so called *configuration files* sometimes also referred to as *bit files* that are uploaded onto the FPGA in order to perform the desired function. The configuration files are obtained using special software provided by the FPGA vendors that synthesize code written in hardware description languages (HDLs) such as VHDL or Verilog HDL into a bit file appropriate for the desired device.

There are two competing methods and technologies for programming FPGAs. First, there is *SRAM programming* where the bit stream is stored into SRAM and then is used to realize combinatorial logic as described in Section 7.2.1. In addition, some of the bits are used to configure the multiplexers that are used to control the special functions and the routing inside the ALM as well as the routing of the interconnect. The advantage of this method is, that it allows the reconfiguration of the FPGA and even makes dynamic reconfiguration possible. But then again, as SRAM is not static the bit stream has to be uploaded on every startup. Typically the bit files are stored in Electrically Erasable Programmable ROM (EEPROM) or flash memory and loaded to the FPGA using a CPLD[3]. In order to protect the intellectual property from being copied, by intercepting the bit stream during configuration of the FPGA, modern devices typically feature an encryption of the bit stream. Therefore, they contain a non-volatile key memory that holds a secret key. The vendor's synthesizer software then generates an encrypted bit stream the same key. Finally, on configuration, the FPGA decrypts the bit stream and configures itself.

The Second method is *anti-fuse programming* where physical connections between the traces are established by applying a current. Anti-fuse based FPGAs have the advantage of being more power efficient and of a better intellectual property security. However, they can only be programmed once, which makes them much more inflexible.

[3] Today, most CPLDs are electrically programmable and erasable and non-volatile.

Chapter 8

Overview of the Hardware Platform

The contribution of this work is based on an existing, specially designed FPGA-based hardware platform and an existing authenticated encryption implementation that is part of the QCrypt project [85], which is evaluated by the Swiss National Science Foundation and financed by the Swiss Confederation via Nano-Tera.ch. The whole system was designed to provide a future-proof, secure, high-speed communication platform that is based on a quantum key-distribution system. This section aims at describing the existing hardware platform and its most important components: the *quantum key-distribution system* and the FPGA-based *fast-encryptor system*. Section 8.1 gives an overview over the overall architecture of QCrypt and its main components. The quantum key-distribution system is explained in Section 8.2 and Section 8.3 describes the fast-encryptor system. Finally, Section 8.4 will introduce the *Altera Stratix IV FPGA*, implementing the fast-encryption system.

8.1 System Overview

Within the scope of the QCrypt project, a dedicated hardware platform, based on an FPGA has been developed to have a real-life test environment. The whole system can be divided into two parts: the *quantum key-distribution (QKD) system* and the *fast-encryptor system* as shown in Figure 8.1. Both communication partners Alice and Bob own such a system. In order to provide a secure key distribution, the *QKD-system* generates truly random numbers using quantum effects and then performs a quantum key distribution over a secure dark-fibre channel using the principles described in Section 3.3 and Section 4. The *fast-encryptor system* handles the high-speed network communication and generates an authenticated and encrypted data stream using the user-keys provided by the QKD-system that is then transmitted to the communication partner over a public 100 Gbit/s channel.

A description of the quantum key-distribution system is given in Section 8.2 and details of the fast-encryptor system follows in Section 8.3.

8.2 The Quantum Key-Distribution System

The *quantum key-distribution (QKD) system* is responsible for the user-key generation and distribution and is described in [105]. Both communication partners Alice and Bob own a *quantum key-distribution board*. The boards are directly connected over a secure 2.5 Gbit/s

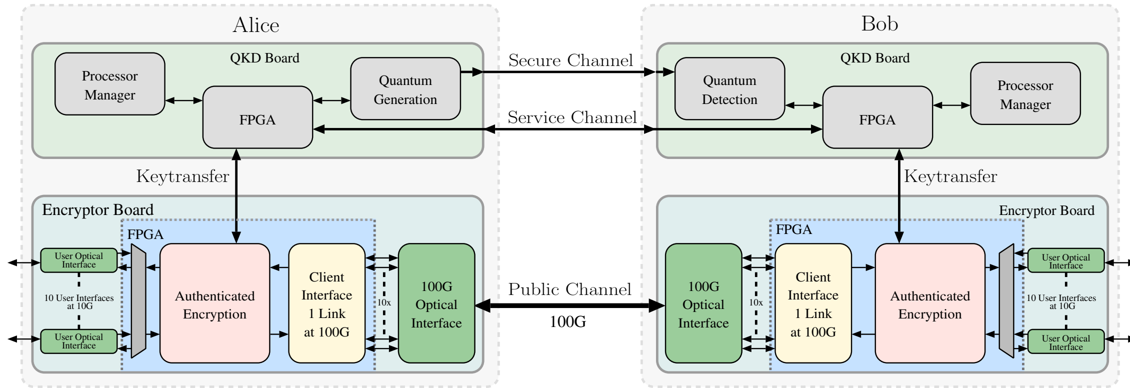


Figure 8.1: Overview of the QCRYPT system.

dark fibre quantum channel and a 2.5 Gbit/s classical service channel. The QKD boards contain an FPGA that is responsible for handling the communication between the QKD boards during key generation as well as for the fundamental operations in the process of quantum key-stream generation. Additionally, the FPGA handles the error correction and privacy amplification that are required in the process of key distillation and uses the service channel for this task. Furthermore, the FPGA passes the user keys over to the process manager and also to the fast-encryptor board if required. The QKD board also features a quantum generation component that is responsible for generating truly random numbers exploiting an optical quantum process [88] which are then used for the transformation into a quantum bit stream following the principles described in Section 3.3. In addition, the board includes a processor manager that handles high-level controls and the user-key management.

Using this setup, random keys can be generated and securely transmitted at 1 Mibit/s.

8.3 The Fast-Encryptor System

The *fast-encryptor system* takes the user data, encrypts and authenticates this data if required and transmits it to the communication partner over a public 100 Gbit/s optical channel. Figure 8.2 shows the actual fast-encryptor board which is based on a 24-layer printed circuit board (PCB). It features ten 10 Gbit/s ethernet user interfaces and one optical 100 Gbit/s as well as an *Altera Stratix IV EP4S100G5 FPGA* that provides around 530,000 logic elements (LEs) and 48 full-duplex clock data recovery (CDR)-based transceivers at up to 11.3 Gbit/s. On startup an *Altera MAX II CPLD* loads the configuration file from a flash memory and configures the FPGA accordingly.

On the FPGA a high-speed authenticated-encryption system that is capable of providing a throughput of 100 Gbit/s is implemented. It consists of a transceiver that is able to exchange plaintext user data over a 10 × 10 Gbit/s ethernet interfaces. When acting as a transmitter, this data is then passed over to the *high-speed authenticated-encryption unit* that provides the possibility to either pass the data through as plaintext, to only encrypt the data or to authenticate and encrypt the data. The implementation of the high-speed authenticated-encryption unit is the main contribution of this work an will be treated in detail in the following chapters. The output of the high-speed authenticated-encryption unit is handed over to the 100 Gbit/s client interface and finally transmitted

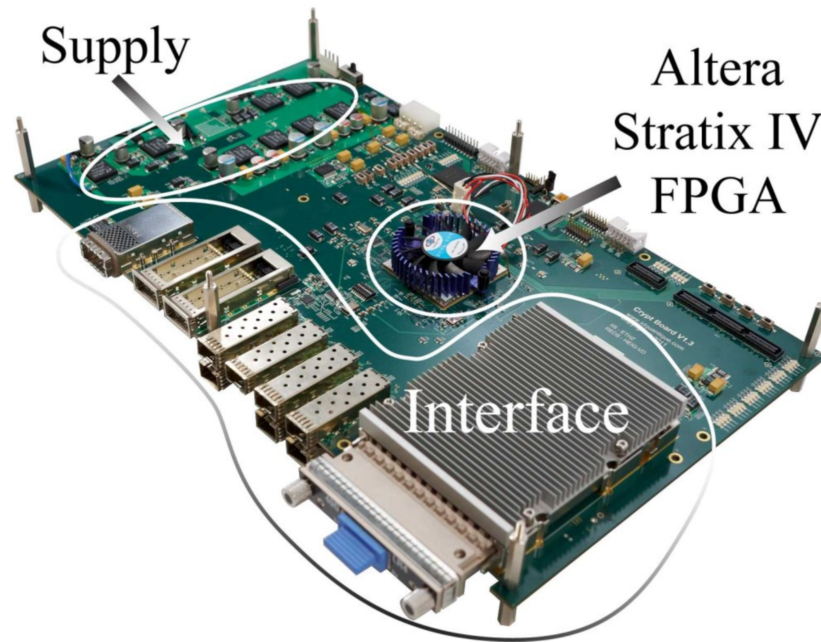


Figure 8.2: Encryption board.

to the communication partner over a 100 Gbit/s public optical channel. The receiver side has a similar system, but with the data taking the reverse way. From the 100 Gbit/s client interface it is passed over to the high-speed authenticated-encryption unit where the data is decrypted and its authenticity is checked if needed. Then it is given to the transceiver that hands the received plaintext data over to the user.

In addition, the board comes with a sophisticated power supply that supports the boards power consumption of up to 340 W and that provides all the voltage levels needed by the components.

8.4 The Altera Stratix IV EP4S100G5 FPGA

The *Altera Stratix IV EP4S100G5 FPGA* belongs to the *Altera Stratix IV GT FPGA* family which is a high-end device class that includes high-speed transceiver features. The EP4S100G5 device has a ball grid array package with 1,932 pins (there of more than 1,100 pins for power and 781 pins for user I/Os) and features 212,480 arithmetic logic modules (ALMs) and a total of 27,376 kbit of memory. Additionally, it provides 32 transceiver channels that support data rates up to 11.3 Gbit/s and 16 transceiver channels that support data rates up to 6.5 Gbit/s.

The following subsections introduce the basic architecture of the Altera Stratix IV EP4S100G5 FPGA and point out the most important characteristics of this device. For a more detailed information the reader is referred to the datasheet [9] provided by Altera.

8.4.1 Logic Array Blocks (LABs) and Adaptive Logic Modules (ALMs)

Logic array blocks (LABs) and *adaptive logic modules (ALMs)* are the fundamental building blocks in the Stratix® IV device family that can be configured to implement logic functions, arithmetic functions and register functions. Each LAB is composed of ten ALMs, various

carry and shared arithmetic chains, control signals, local interconnect and register chain connection lines [9]. The local interconnect links the ALMs inside the same LAB. The direct link interconnect connects to the local interconnect of the neighboring LABs to its left and to its right. Register chain connections allow to transfer the output of an ALM to the subsequent register.

In addition, a variation of the Stratix IV LABs called memory LAB (MLAB) exists which adds look-up table based SRAM capability to the LAB and which supports a maximum of 640 bits of simple dual-port SRAM per MLAB. In the Stratix IV family LABs and MLABs always exists as pairs.

The *adaptive logic module* as outlined in Figure 8.3 is the fundamental unit for building logic in the Stratix IV FPGA. Each ALM contains a number of LUT-based ressources that can be split between two adaptive LUTs (ALUTs) and two registers. Using up to eight inputs the two ALUTs can implement various combinations of two logic functions with the additional option for each ALM to implement any six-input function and even certain seven-input functions.

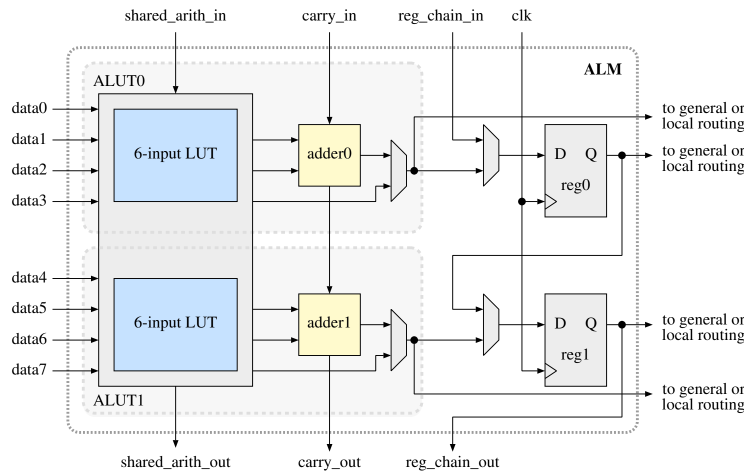


Figure 8.3: High-level block diagram of the Stratix IV ALM.

Additionally, each ALM features two programmable registers and two dedicated full adders. Using these dedicated ressources and the provided configuration and interconnect possibilities it is possible to efficiently realize various arithmetic functions and shift registers.

Each ALM can be used in one of the following operating modes that uses the ALM's resources differently. The *normal mode* is suited for general combinatorial and logic functions and allows different input combinations for the two LUTs reaching from two 4-input functions, to combinations like one 5-input functions and one 3-input function and even to two 6-input functions in special cases. The *extended LUT mode* can be used to implement a special set of 7-input functions. This set requires to include a 2-to-1 multiplexer that is fed by two five-input functions that have four inputs in common. Such functions often appear in "if-else" statements in HDL code. The *arithmetic mode* is perfectly suited for implementing arithmetic functions such as adders, counters, comparators, accumulators, and wide parity functions. Each ALM contains two dedicated full adders that can add the outputs of two 4-input logic functions implemented in the LUTs while also considering the carry in bit. In the *shared arithmetic mode* each ALM can realize a 3-input add and in the *LUT-register mode* the ALM's LUTs can be used to add a third register to the two registers included in the ALM.

Chapter 9

The Existing Authenticated-Encryption Engine

This chapter describes the existing authenticated-encryption engine which features two completely independent AE cores. The architecture of the AE cores is based on the AES block cipher which is used in the Galois Counter Mode of operation and follows the design described by Henzen and Fichter in [49]. As shown in Figure 9.1, one AE core is dedicated to encryption and the other core is solely used for decryption. Both AE cores can use distinct keying material as delivered by the QKD board and are able to work separately from each other.

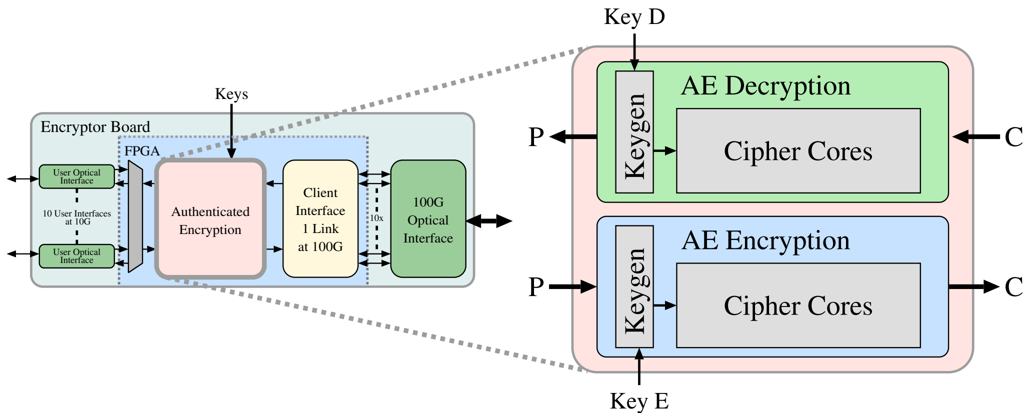


Figure 9.1: Overview of the authenticated-encryption engine.

The GCM system can be divided into two main components. First, the *multi-core AES encryption* block which is responsible for encryption and second, the *parallel pipelined GHASH* block which provides authentication. The multi-core AES encryption architecture is described in Section 9.1 and the GHASH design is treated in Section 9.2. Finally, the overall *GCM* architecture is outlined in Section 9.3.

9.1 Multi-Core AES Design

In order to provide a high throughput, pipeline stages are introduced after each round of AES. The most critical transformation within a round of AES is the *SubBytes* procedure which includes the use of 8-bit S-boxes. As these S-boxes are implemented using BRAM which is configured as registered ROM, this introduces another in-round pipeline stage. Furthermore, the design includes an input register. This results in $2 \times 10 + 1 = 21$ pipeline stages for AES with 128-bit keys.

Still, a single AES core is not capable of providing a throughput of 100 Gbit/s. Consequently, in order to reach the requested throughput four instances of the AES core are used in parallel. All the cores share the same key scheduler which results in an architecture for the multi-core AES module as shown in Figure 9.2. As they share the key scheduler the AES cores are forced to use the same user key (and thus also the same round keys). The resulting multi-core design is capable of processing a 512-bit block (4×128 bit) of plaintext each cycle without delay after prior initialization. This is possible since the GCM uses the block cipher in Counter (CTR) mode of operation where the cipher text is generated by XORing the plaintext with the output of the block cipher as described in Section 6.5.

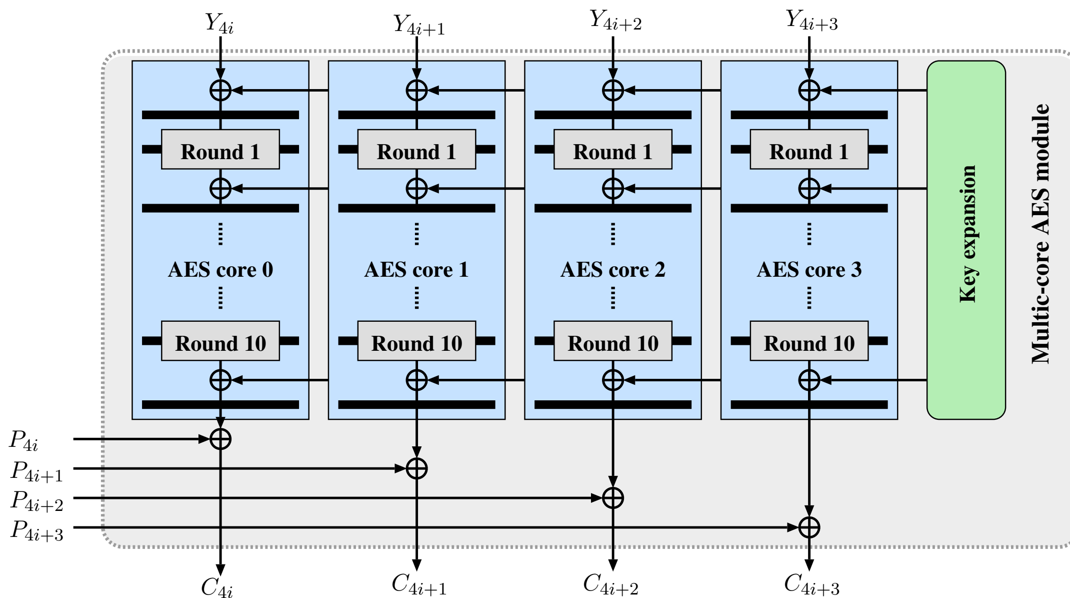


Figure 9.2: Overview of the multi-core AES architecture. Connections are 128 bit wide and $i = 0, 1, \ldots, \lceil \frac{n}{4} \rceil$ where n is the number of message blocks.

9.2 The Parallel Pipelined GHASH Design

In order to be able to work with the multi-core AES, the authentication core is based on four parallel pipelined binary-field multipliers. As the GHASH is the most complex component of the entire GCM architecture it requires some optimization. To allow the parallelization of the multiplication operation described in Algorithm (6.3) an approach as outlined in [96] is used. The goal is to divide the sequential multiplications and addition steps of Algorithm (6.3) into parallel multiplications that produce the same output. Following [96] and by defining a parallelization degree q, the calculation of the last output block X_{m+n+1}

in Algorithm (6.3) can be rewritten as a sum of q sub-terms Q_i:

$$X_{m+n+1} = Q_q \oplus Q_{q-1} \oplus \cdots \oplus Q_1 \tag{9.1}$$

where

$$\begin{aligned} Q_q &= (((I_1 H^q \oplus I_{q+1})H^q \oplus I_{2q+1})H^q \oplus \dots)H^q \\ Q_{q-1} &= (((I_2 H^q \oplus I_{q+2})H^q \oplus I_{2q+2})H^q \oplus \dots)H^{q-1} \\ &\vdots \\ Q_1 &= (((I_q H^q \oplus I_{2q})H^q \oplus I_{3q})H^q \oplus \dots)H \end{aligned} \tag{9.2}$$

and

$$I_1, I_2, \dots, I_{m+n+1} = (A_1, \dots, A_m^* \| 0^{128-v}, C_1, \dots, C_n^* \| 0^{128-u}, \text{len}(A) \| \text{len}(C)) \tag{9.3}$$

In case $m+n+1$ is not a multiple of q, the last q multiplications are shifted appropriately. The important observation of Equation (9.1) is the fact, that the different Q_i can be calculated separately and XORed at the very end of the GHASH operation.

Still, because of the high-speed requirement additional pipeline stages had to be introduced in every multiplier which results in a design that uses the 2-step Karatsuba-Ofman algorithm (KOA) and a 4-stage pipeline architecture similar to [113]. The KOA aims at reducing the complexity of a multiplication while simultaneously enabling the introduction of pipelining. The single step KOA splits two m-bit operands A and B into four $\frac{m}{2}$-bit terms A_l, A_h, B_l and B_h (*split* phase). The multiplication works as follows:

$$\begin{aligned} R_l &= A_l B_l \\ R_{hl} &= (A_h + A_l)(B_h + B_l) \\ R_h &= A_h B_h \\ R &= R_h x^m + x^{\frac{m}{2}}(R_h + R_{hl} + R_l) + R_l \end{aligned} \tag{9.4}$$

where x is the base of the used number representation. The final result R is calculated by aligning the intermediate results R_h, R_{hl} and R_l (*alignment* phase). If this approach is applied recursively twice, it is referred to as 2-step KOA. Using the 2-step KOA each 128-bit multiplier is reduced to nine 32-bit multipliers which decreases the complexity of the computation. In addition four pipeline stages were inserted to each 2-step Karatsuba-Ofman multiplier which results in an architecture as shown in Figure 9.3.

As four parallelization degrees and four pipeline stages are used the parallelization degree q has to be set to 16 which means that input blocks I_i are processed into 16 separate registers Q_i with $1 \leq i \leq 16$. Figure 9.4 gives a block diagram of the final GHASH module. The multiplexers select the first operand for the multiplication while the second operand H^k with $1 \leq k \leq 16$ comes from pre-calculated values that are stored in a dedicated memory bank called H^k-memory. The authentication tag T is then calculated by XORing all 16 values of $Q_1, \dots, Q_{16}$.

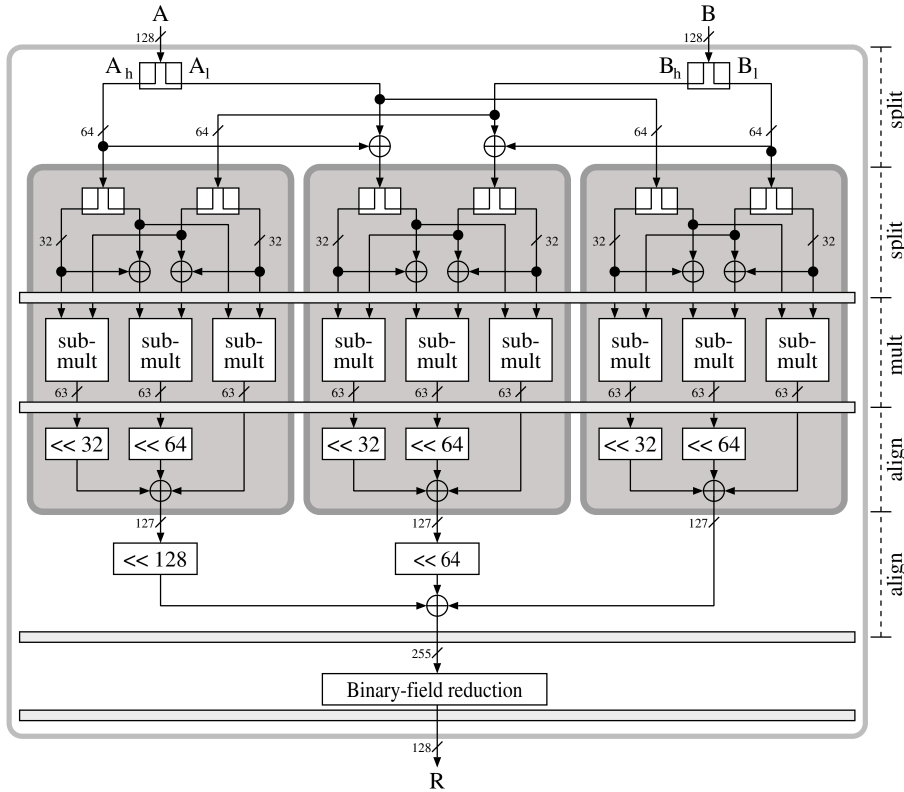


Figure 9.3: Architecture of the 2-step Karatsuba-Ofman multiplier including four pipeline stages.

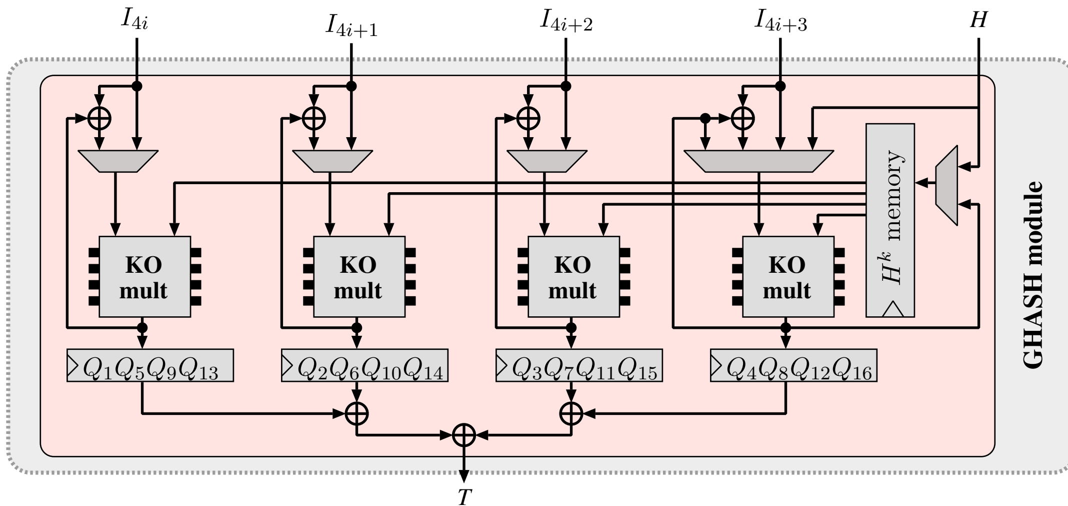


Figure 9.4: Overview of the parallel pipelined GHASH architecture. The four KO-mult modules represent 2-step KOA multipliers. Connections are 128 bit wide and $i = 0, 1, \ldots, \lceil \frac{m+n+1}{4} \rceil$ where n is the number of message blocks.

9.3 GCM Design

The overall GCM architecture is based on a combination of the multi-core AES as described in Section 9.1 and the parallel pipelined GHASH as outlined in Section 9.2. The powers of H are computed whenever the secret key changes. After the calculation of the initial H-value according to Algorithm (6.1), which is performed by encrypting the zero message 0^{128}, 17 cycles are required to calculate all 16 powers of H and to store it in the dedicated H^k-memory. After this step the AES-GCM core is ready to authenticate and encrypt messages with the new key.

In Algorithm (9.1), the calculation of the accumulators Q_i involves a multiplication with different powers of H. Thus, the GCM controller needs a priori knowledge of the overall message length. While most of the input blocks are multiplied with H^{16}, the last 15 blocks involve multiplications with lower powers of H. Table 9.1 shows the input pairs of the four parallel multipliers for an input of 288 bytes of ciphertext with 48 bytes of additional authenticated data. Note, that for this example the number of 128-bit input blocks is $m + n + 1 = 3 + 18 + 1 = 22$ is not a multiple of the chosen parallelization degree $q = 16$.

Table 9.1: Input pairs of the four parallel multipliers for an input of 288 bytes of ciphertext with 48 bytes of additional authenticated data.

Clk	I_{4j+1}	H^k	I_{4j+2}	H^k	I_{4j+3}	H^k	I_{4j+4}	H^k
1	A_1	H^{16}	A_2	H^{16}	A_3	H^{16}	C_1	H^{16}
2	C_2	H^{16}	C_3	H^{16}	C_4	H^{16}	C_5	H^{15}
3	C_6	H^{14}	C_7	H^{13}	C_8	H^{12}	C_9	H^{11}
4	C_{10}	H^{10}	C_{11}	H^9	C_{12}	H^8	C_{13}	H^7
5	C_{14}	H^6	C_{15}	H^5	C_{16}	H^4	C_{17}	H^3
6	C_{18}	H^2	len[a]	H				

[a] len = len(A)||len(C)

To be able to use the correct values of H^k the controller needs to know at the second clock cycle, that C_5 has to be multiplied with H^{15} and that it needs to scale the powers of H^k correspondingly for the next four clock cycles. The problem of knowing this in advance is solved by inserting a 4-stage 512-bit buffer at the input of the AES-GCM core, which results in an overall architecture as presented in Figure 9.5. This buffer is used as a shift

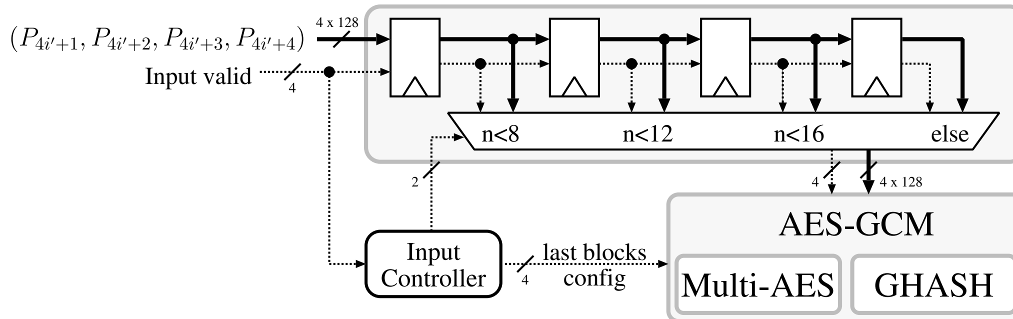


Figure 9.5: Overview of the AES-GCM input-buffer architecture. Solid connections are 512 bit wide, dotted connections are 4 bit wide and $i' = 0, 1, \ldots, \lceil \frac{n}{4} \rceil$.

register and allows the controller to detect the length and the configuration of an input message in advance. The multiplexer, following the input buffer is used to select the correct input in case of short messages where $m + n + 1 < 16$. So, by the cost of introducing four 512-bit registers and a 512-bit 4:1 multiplexer, the GCM core is able to process messages without prior information of the overall message length.

GCM has the advantage, that encryption and decryption work the same way. On the one hand, the underlying block cipher does not need to implement decryption functionality and on the other hand, the same architecture can be used for encryption and decryption. There are only two minor changes necessary. First, the ciphertext can be directly applied to the GHASH core as the GHASH expects either encrypted or authentication-only inputs. Second, the decryption architecture has to check the validity of the resulting authentication tag by comparing it with the original tag. In fact, the existing implementation is capable of encryption and decryption, although this is not required by the overall authenticated encryption engine.

Chapter 10

Analysis of Alternatives to the Existing System

One of the main goals of this work is to find and implement alternatives to the block cipher and the authenticated encryption engine of the existing system in order to provide quick fixes in case of a vulnerability of one of the used components. In the best case these alternatives should even improve the possible throughput or the required FPGA resources (or preferably both) compared to the existing system.

In order to find appropriate candidates, first, in Section 10.1 an analysis of the characteristics and conditions these candidates need to fulfill to meet the requirements is performed. Then, in Section 10.2 possible candidates as alternative block ciphers are inspected in regard of their suitability with the previously defined characteristics and requirements. The same approach is then exercised in Section 10.3 to find and analyze possible alternatives for the authenticated-encryption engine.

10.1 Analysis of Requirements

This section defines the requirements and characteristics of the high-speed authenticated encryption system. Furthermore, it elaborates how the system's characteristics influence the choice of properties that are desired for possible alternatives to the existing system.

The following requirements can be identified, which will be discussed in the following:

- 100 Gbit/s authenticated encryption with associated data
- Compatible to existing authenticated-encryption engine
- Different modes of transmitting a frame: leave it unchanged, authenticate and encrypt the whole frame, authenticate and encrypt the payload, only encrypt the payload or only authenticate the payload
- Low latency
- Low setup time after key change
- Short delay between ciphertext and authentication tag
- Low transmission overhead caused by AE

- (Preferably low hardware consumption)

As previously stated, the main objective is to provide authenticated encryption at a speed of 100 Gbit/s as supported by IEEE 802.3ba ethernet standard [52]. Basically, there are two main principles to realize a high throughput: *pipelining* in combination with *loop unrolling* and *parallelization*. The concept of *pipelining* aims at increasing the throughput by cutting the combinational logic function f into p separate stages of reduced and approximately uniform computational delays by inserting registers in between. This effectively reduces the longest path delay t_{lp} to $t_{lp}(p) \approx \frac{t_f}{p} + t_{reg}$, where t_f is the delay of the combinational function and t_{reg} is the register delay, and thus allows higher operating frequencies [56]. At the same time it also increases the latency by the number of pipeline stage p and requires p additional register banks. In order to fully benefit from pipelining it is necessary to unroll functions f that use feedback loops as the round functions of block ciphers typically do. Again, this increases the hardware consumption by $f \cdot r$ where r is the number of loops being unrolled. The concept of *parallelization* is a form of replication by simply providing q instances of the same functional units for f and letting them process independent inputs. The throughput as well as the hardware consumption is roughly multiplied by q. Note, that it is strictly necessary that the inputs to the functional units are independent from each other to allow parallelization.

The theoretical maximum throughput of a system applying pipelining techniques is only achieved if the pipeline is completely filled. Only then a valid output is produced every cycle. Still, every input applied to the input of f has a latency of p cycles to appear at the output. Thus, it is advantageous to have a minimal number of pipeline stages p and to avoid pipeline stalls. Two important cases that can cause pipeline stalls in most AE schemes are changes of the secret key K and changes of the initial vector IV as they usually include an initialization phase. Thus, the number of pipeline stalls caused by these two operations should be as small as possible.

As elucidated in Chapter 9, the existing authenticated encryption engine is based on four AES-128 cores used in parallel. Consequently, the network interface is designed to deliver $4 \cdot 128 = 512$ bits per cycle to the AE engine. Thus, in order to be compatible to the existing system and to allow an easy exchange of the AE engine, alternatives should accept 512 bits per cycle as well. Furthermore, the existing AE engine uses 128-bit secret keys received from the QKD system. This fact has to be considered by alternative candidates.

In addition, different modes of encrypting and transmitting ethernet frames are desired. First, it should be possible to leave an ethernet frame unchanged which can easily be done by simply bypassing the authenticated encryption engine. The second option is to authenticate and encrypt a whole ethernet frame and to encapsulate it into a new frame. This principle is shown in Figure 10.1a. Here the entire ethernet frame is provided as plaintext for the AE engine. The initial Vector IV, the encrypted frame i.e. the resulting ciphertext C and the authentication tag T are encapsulated by the internal start of frame (SOF) and the internal end of frame (EOF). The resulting frame is then transmitted to the communication partner. Note, that this mode is only applicable in point-to-point network configurations as the encryption of the header hides the address information needed for routing. It provides a high level of privacy for frames since in addition to the payload also the source and destination addresses are hidden. Still, it has a comparably high transmission overhead of $g_s + g_e$ where g_s includes the internal SOF and the IV and g_e includes T and the internal SOF.

A third option, as shown in Figure 10.1b, is to authenticate and encrypt only the

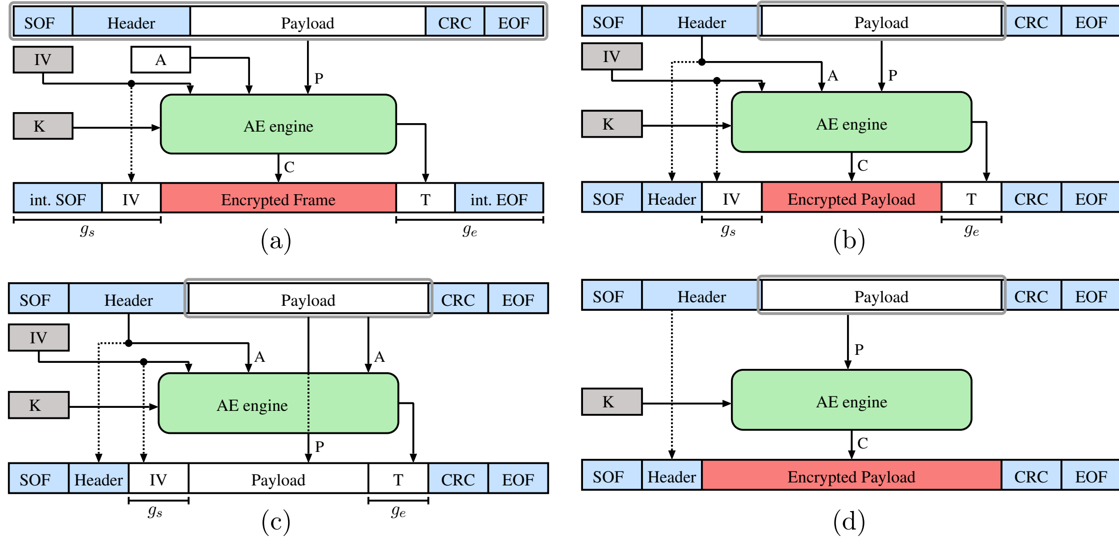


Figure 10.1: Different modes of encrypting an ethernet frame.

payload of a frame and to authenticate its header. This method applies the AEAD scheme. The new frame is composed of the SOF, the original header, the encrypted payload, the calculated authentication tag T, a cyclic redundancy checksum (CRC) which should be recalculated for the new frame and the EOF. The transmission overhead is made up of the initial vector IV and the authentication tag T. As the header is sent in plaintext, this method is also suitable for multi-point networks. It is also possible to only authenticate header and payload as outlined in Figure 10.1c. However, this option does not provide any privacy for the transmitted data and is not major importance. Again, the transmission overhead is made up of the IV and T. Finally, one option is to only encrypt the payload of the frame. In consequence, this method does not provide authenticity for the frame. As no IV and authentication tag are required there is no transmission overhead compared to a standard ethernet frame.

As visible in Figure 10.1, some transmission modes require the cipher text C as well as the authentication tag T to compose the ethernet frame that is then transmitted over the public channel. If the generation of T is delayed for n cycles after the appearance of the last block of the cipher text C, C has to be preserved for n cycles. This requires $n \cdot (\text{len}(\text{frame}) - \text{len}(T))$ bits of additional memory, where $len()$ returns the number of bits of its argument. Consequently, it is essential to have a short delay between ciphertext and authentication tag.

A low hardware consumption is not the most important requirement as long as the required resources do not exceed the resources left on the target FPGA. The, network interface components and controllers require about 120 kALMs which is about 56% of the resources available on the Altera Stratix IV EP4S100G5 FPGA. So, about 90 kALMs are available for the AE-engine. Still, a low hardware consumption is desirable as it leaves room for future feature extensions and as a lower resource consumption typically allows higher frequencies due to a lower routing overhead.

In Section 10.2 and Section 10.3 block ciphers and authenticated encryption systems are surveyed in respect to the requirements discussed in this section.

10.2 Alternative Block Cipher

The existing authenticated-encryption engine is based on AES-GCM which is a block-cipher mode of operation. Therefore, in order to provide an alternative in case the security of AES should be compromised or to possibly find an alternative that is better suited for the system than AES, we investigate into alternative block ciphers. This section surveys how the previously described requirements relate to general properties of block ciphers and will then analyze and compare promising block ciphers.

As mentioned in Chapter 9, the existing system uses four AES cores in a parallel. So, to facilitate the exchange of the encryption unit of the existing GCM implementation an alternative authenticated-encryption unit should be able to reach the desired throughput of 100 Gbit/s using four instances of the cipher core. This results in a target throughput of 25 Gbit/s for a single block cipher core. The maximum throughput of a block cipher is mainly determined by the operations it uses and how efficient they can be realized in hardware. Functions like shifts or rotations come for free and basic logic functions such as XOR, AND, OR, NAND or NOR can efficiently be implemented in hardware. Mathematical operations such as additions or multiplications are typically more expensive. Another important point is the number of round functions of the block cipher, or more precisely the number of pipeline stages required to accomplish the frequency needed to reach the desired throughput of 100 Gbit/s. As pointed out, the number of pipeline stages directly relates to the number of cycles of latency. A low latency is desirable because it moderates the negative effect of pipeline stalls. Consequently, the fewer pipeline stages necessary, the better. It is also desirable that key changes imply a minimal number of pipeline stalls. Many block ciphers require the round-keys to be applied in reverse order. Consequently, the encryption can only start when the last round key has been computed by the round-key generator. So if the calculation of the round keys needs n clock cycles this causes n pipeline stalls.

As stated in Chapter 9 the authenticated-encryption engine employs two separate AE cores. One core is exclusively dedicated to encryption and the other one exclusively performs the decryption. Thus, in case the AE algorithm requires the block cipher in encryption and decryption configuration, both configurations should be able to reach the target throughput. Even if it is not of major interest whether the encryption and the decryption of a block cipher are able to share functionality and hardware it is desirable as this could be beneficial in some cases.

In the following, we analyze the characteristics and the suitability for a throughput of 100 Gbit/s for two block ciphers. First we treat the currently used AES cipher and then investigate into Serpent cipher. Finally, we will compare both ciphers to each other and select an alternative block cipher based on the observations and findings of this comparison.

10.2.1 AES

The AES block cipher as described in Section 5.1 has 10 rounds if used with a key size of 128 bits. Each (but the very last, which is slightly different) round includes the *SubBytes*, the *ShiftRows*, the *MixColumns* and the *AddRoundKey* operations.

The *SubBytes* operation is based on S-Boxes that perform an 8-bit to 8-bit mapping. There are three methods to realize the SubBytes operation: *algorithmic*, using *LUTs* or using *RAM*. The algorithmic approach directly implements the multiplicative inverse and the affine transform in combinational logic. However, then a number of pipeline stages

has to be introduced to allow the required operating frequencies. This would increase the latency to an unacceptable level. Implementing the S-Boxes in LUTs requires 4 ALMs and two levels of cascaded multiplexers (MUX) on the target platform, as each ALM can be configured as a 6-to-1 LUT [65]. This creates three logic levels which eventually require the insertion of a pipeline stage which would still be acceptable. Using RAM to implement the S-Boxes is the most common approach. Altera Stratix IV FPGAs provide to either use BRAM or to configure MLABs as ROM. Both cases provide the possibility to introduce pipeline stages for free if required. The *ShiftRows* operation is for free as it only requires rewiring. For the encryption the *MixColumns* operation can be implemented with relatively few resources as it only includes multiplications in $GF(2^8)$ with the constant values `0x01`, `0x02`, and `0x03`. These can be performed using shifts and XORs in hardware. However, the *InvMixColums* operation required for the decryption includes multiplications in $GF(2^8)$ with the constants `0x09`, `0x0b`, `0x0e` and `0x0d`. These are more complex to implement and tend to form a longer critical path. Hence, the maximum frequency for decryption may slightly be lower than for encryption. The *AddRoundKey* operation is cheap in hardware and only requires to XOR two 128-bit values. While in theory AES encryption and decryption can share some of their resources, resource sharing is not especially practical for high speed implementations, as it requires the introduction of additional multiplexers which would increase the critical path and thus lower the maximum operating frequency.

Each round of AES requires one round key of 128-bit. The round-key generation is relatively cheap and has a short critical path, as it only requires fixed rotates, XORs and four 8-bit to 8-bit S-Boxes. As it can roughly reach the same frequency as the encryption operation, this allows to compute the round keys on-the-fly without lowering the overall maximum frequency.

Existing high-speed AES implementations on FPGAs [6, 28, 49] suggest, that two pipeline stages per round are necessary to reach the desired throughput of 25 Gbit/s for one cipher core. Including an input buffer this results in a total of 21 pipeline stages and consequently a latency of 21 clock cycles. The existing AES architecture, combining one round-key generation unit and four cipher cores, requires around 6,916 ALMs and 314 M9K BRAM blocks for encryption and 6,964 ALMs and 314 M9K BRAM blocks for decryption. It reaches a throughput of 124 Gbit/s at a maximum frequency of 252 MHz. The latency is 21 clock cycles and 20 cycles are required to compute all round keys.

10.2.2 Serpent

The Serpent block cipher as outlined in Section 5.2 has 32 rounds, requires the computation of 33 round keys and has a high security level. Each round (but the very last, which is slightly different), is composed out of the *key-mixing stage*, the *substitution stage* and the *avalanche stage*.

The *key-mixing stage* is cheap in hardware as it only requires to XOR two 128-bit values. The *substitution stage* uses eight different S-boxes that perform a 4-bit to 4-bit substitution. These 4-bit to 4-bit substitutions fit very well to the structure of the Alter Stratix IV ALMs as these can be configured to provide two 4-to-1 LUTs. So, one S-box can be implemented using only 2 ALMs, which results in $2 \times 32 = 64$ ALMs required for the S-Boxes of one round of Serpent and $64 \times 32 = 2{,}048$ ALMs for all 32 rounds. The *avalance stage* or *linear transformation* only uses fixed shifts and rotations, which are free in hardware, and two levels of XORs. Thus, the critical path is only increased by the delay of two XORs. Serpent encryption and decryption can only share few resources. As resource sharing requires the

introduction of additional multiplexers which would increase the critical path and thus lower the maximum operating frequency while only providing marginally lower resource consumption it is impracticable for high-speed implementations.

The key-scheduler that computes the 33 round keys is relatively complex. On the one hand it requires all eight different S-Boxes, which requires $64 \times 8 = 512$ ALMs and on the other hand eight levels of XORs per round key which in fact constitutes the critical path of the Serpent ciphers.

Existing implementations [33, 38, 39, 64, 103] of the Serpent block cipher indicate that Serpent is well suited for high speed implementations and can reach a throughput of up to 40 Gbit/s. Using one pipeline stage should be sufficient to reach the target throughput of 25 Gbit/s for one cipher core. Due to that fact that Serpent uses 32 rounds this results in a latency of 33 clock cycles if an additional input buffer is used. As it employs functions that are cheap in hardware, the maximum frequency should be relatively high. However, the big number of rounds and pipeline stages, as well as the complex key scheduler result in a fairly high resource consumption.

10.2.3 Comparison and Choice of an Alternative Blockcipher

Table 10.1 compares the characteristics of the AES cipher with those of two candidates for the alternative block cipher. Namely, these candidates are *Serpent* and *Twofish* [98]. All three ciphers were finalists in the AES-competition and during the competition process various comparisons of the AES-candidates targeting their hardware performance were published [33, 39, 45]. Still most of the papers try to reach good *Throughput/Area* results and do not really target high-speed implementations. Thus, the values for maximum throughput, maximum frequency, hardware consumption and required pipeline stages had to be estimated for Serpent and Twofish. The values of AES correspond to the AES implementation used in the existing AE engine as described in Section 9.1. Due to advances in FPGA technology, performance enhancements can be expected compared to the papers previously mentioned. Both alternative candidates—Serpent and Twofish—should be able to reach a throughput of 100 Gbit/s using four cipher cores in parallel. A detailed description of the Twofish cipher is out of scope of this theses. We refer the reader to [98] for an algorithm description.

We compare Serpent and Twofish with AES as this cipher is used in the existing implementation. Both, Serpent and Twofish support the required block- and key-length of 128 bits and based on published implementations and own research they should be capable to reach the desired throughput. However when opting for maximum throughput Serpent appears to be the most promising candidate with a potential throughput of up to around 37 Gbit/s as it only uses simple operations such as fixed shifts, fixed rotations and XORs which allow a short critical path. Then again, Serpent, consumes most resources in terms of used ALMs when implementing the S-Boxes as LUTs. However, it does not require the use of any BRAM. Twofish in contrast should consume about as many resources as the existing AES implementation does with a higher need of BRAM. Serpents uses a conservative security approach and has a high security margin and consequently has a high number of rounds. The high number of rounds in combination with the need to apply pipelining results in a latency of 33 clock cycles. This is around 50% higher compared to AES. Twofish seem to require an even higher number of pipeline stages of at least 33 stages but most probably 65 stages to reach the target throughput. The latter would result in an unacceptable latency. Compared to AES, Serpent has a relatively complex round-key

Table 10.1: Overview of block ciphers.

Feature	**AES**	**Serpent**[a]	**Twofish**[b]
Block size	128/192/256	128	128
Key size	128/192/256	up to 256	128/192/256
Enc/Dec sharing	good	limited	full
Approx. f_{max} *[MHz]*	252	290	270
Approx. throughput [Gbit/s]	28.16	37.12	34.56
Approx. area [ALMs][c]	7000	26000	7000
Approx. M9K BRAM blocks[c]	314	0	520
Rounds	10	32	16
Pipeline stages[c]	21	33	32+
Logical functions used	xor $GF(2^8)$ mult	xor	xor $GF(2^8)$ mult add mod(2^{32})

[a] Values approximated from the results in [33, 38, 39, 64, 103].
[b] Values approximated from the results in [33, 39, 45].
[c] For target platform Altera Stratix IV to reach throughput of min. 100 Gbit/s per cipher core.

generation. However, as only one key-generation unit is required for four parallel cipher cores, and the key generation should be able to keep up with the maximum frequency reached by the ciphers cores, this is not a major issue.

Taking all pros and cons into account **we chose to implement Serpent as an alternative to AES**. It is well-analyzed, offers a high security margin and is able to achieve a throughput that even surpasses the throughput of AES. Still, it has the drawback of a higher latency and it consumes more resources compared to AES. In addition, resource sharing between encryption and decryption is not practicable, which can be an issue in certain constellations.

10.3 Alternatives Modes for Authenticated Encryption

This section first elaborates different methods to achieve authenticated encryption and how they comply with the requirements outlined in Section 10.1. After identifying promising methods, concrete algorithms and their attributes are discussed in detail and compared to each other with respect to the requirements. As stated in Chapter 6 there are two basic approaches to achieve AE: the generic composition scheme and integrated authenticated-encryption algorithms.

Among the generic composition scheme the preferred method is the Encrypt-then-MAC approach which involves an encryption step and a MAC-computation step. As discussed in Section 3.2.2 MACs can be constructed using three different approaches: *based on block ciphers*, *based on hash functions* and by defining *customized MACs* which are often based on universal hash functions. To use a block-cipher-based approach is not practicable. In order to reach the required throughput, it would be necessary to use a block cipher as fast as the one used for the encryption. This would effectively more than double the resource consumption and also imply a delay between the generation of the authentication tag T and

the appearance of the last block of the cipher text C that is at least as big as the latency caused by the used block cipher. As previously stated such a big delay is not acceptable as it would drastically increase the memory required for caching. Note, that the generic Encrypt-and-MAC approach could be a solution to the problem of delayed tag calculation for the encryption operation. In the decryption operation, first the ciphertext is decrypted and then the tag is calculated over the plaintext. This again introduces an unwanted delay. Also, while the Encrypt-and-MAC can be secure, it is not generally guaranteed to be secure even if the used encryption and MAC are secure and the components have to be carefully chosen and analyzed [13]. Thus, because of the tag delay for decryption and the security issues we decided not to follow this approach. Using MACs constructed from hash functions suffer from similar disadvantages regarding delayed tag calculation. While state of the art hash functions like Keccak [20, 82] which is the winner of the NIST SHA-3 competition [81] and other SHA-3 competition finalists can reach noticeable throughput [5, 46, 50, 51] the throughput is mainly based on wide input blocks of over 512 bits length and fairly high operating frequencies. This is problematic, as the encryption unit is only capable of delivering 512 bits per clock cycle. In addition, the hash functions are usually based on a fairly high number of rounds. As loop unrolling has to be applied to be able to process an input block every clock cycle, this would also result in a high number of pipeline stages that are required to allow the operating frequencies needed to achieve a throughput of 100 Gbit/s. Again, this would result in an unacceptable delay for the authentication tag. Most customized MACs are based on universal hash functions and work by first compressing the message to be authenticated using the universal hash function and then encrypting the compressed image using a pseudorandom function [8]. In most cases a block cipher is used as a pseudorandom function. Once more, this results in a delayed authentication tag computation which is not practicable. So overall, mainly due to the delayed tag calculation caused by the use of the MAC the generic composition scheme is not suitable as a possible replacement for the existing AE engine.

Another method to realize AE are integrated authenticated-encryption algorithms. The CAESAR challenge yielded many new and promising authenticated-encryption schemes that appear to be auspicious for high-speed implementations. Some of the submissions like ASCON [35] or MORUS [109], are fully parallelizable and only use functions that are cheap to implement in hardware and could prove to be very fast. However, when considering that this work is part of a commercial product, and that the challenge is in an early phase and eventual security flaws in the cipher may not yet be found, we opted for well studied algorithms. These include *Counter with CBC-MAC (CCM)* [53, 77], *EAX* [12, 15], *Carter-Wegman Counter (CWC)* [59] and the *Offset CodeBook (OCB) mode* [61, 62, 63]. The main objective is to be able to perform AEAD with a throughput of 100 Gbit/s. As already stated, it is necessary that the alternative AE mode can be parallelized in order to be able to reach the target throughput. So, as the CCM mode and the EAX mode are not parallelizable they are not suited to achieve the desired throughput and are thus not included in the following analysis. An alternative mode for AE should have a minimal delay of the tag calculation due to the reasons discussed earlier. In addition, a high key agility is desired. Typically, a key change requires the AE algorithm to be reinitialized. This often includes the calculation of variables that require calls to the block cipher which causes consecutive pipeline stalls. Consequently, the number of pipeline stalls caused by a key change should be as low as possible. The very same is true for new massage. The initialization phase required to be able to authenticate and encrypt a new message should need as few clock cycles as possible. It is also important, that the AE algorithm is online

as the network interface is not able to tell the overall message length in advance.

10.3.1 GCM

Galois Counter Mode of Operation as outlined in Section 6.5 can be divided into two parts: the *Encryption part* and the *Authentication part*. It provides AEAD and incremental MAC and is provable secure.

The *encryption part* uses the underlying block cipher in CTR mode. This has two advantages for hardware implementations: first, it is parallelizable, and second the underlying block cipher is only required to provide encryption capability. The *authentication part* is the critical part of GCM. It makes use of a universal hash function called GHASH which uses multiplications in $GF(2^{128})$. As described in Section 9.2 these multiplications can be parallelized and pipelined which results in a delayed tag calculation of 4 clock cycles.

The initialization after a key change requires one block cipher call and additional 17 cycles to precalculate the powers of H required for the parallel GHASH design (see Section 9.2 and Section 9.3 for details). However, as the encryption part of GCM can continue to encrypt during calculation of the powers of H, the GCM core is ready to authenticate and encrypt as soon as all H values are computed and stored. In addition each message requires the use of a new IV, which has to be padded an encrypted. This requires one block cipher call but only causes a latency of one cycle but no pipeline stalls as messages can be applied right after the IV. In addition four input buffers had to be introduced, causing another four cycles of latency.

GCM allows different authentication tag lengths and only expands the input message to a multiple of the block size.

The existing GCM implementation excluding the block-cipher cores requires around 17,000 ALMs and reaches a throughput of 105 Gbit/s at a maximum frequency of 206 MHz. For the entire engine, including two GCM cores, one for encryption and one for decryption this results in a resource consumption of about $2 \times 17{,}000 = 34{,}000$ ALMs.

10.3.2 OCB

The Offset CodeBook mode as described in Section 6.6 is a single-pass scheme that provides AEAD. It uses computations that can efficiently be implemented in hardware. Encryption and decryption have slightly different properties in OCB. In both cases the initialization phase after a user key change requires one block cipher call. The pipeline is stalled until the resulting ciphertext is computed. Each input message requires the use of a new IV. If this IV is a counter, every 64th message requires an additional block cipher call that causes pipeline stalls. In addition, to complete the computation of the authentication tag another block-cipher call is needed. For encryption this block-cipher call does not introduce any pipeline stalls. However, for decryption this final block-cipher call can only be performed after the very last plaintext has been computed. This introduces l pipeline stalls, where l is the latency of the underlying block cipher and causes a delay of the authentication tag calculation of l cycles. In addition, OCB decryption needs encryption and decryption functionality for at least one underlying block cipher core. This, increases the overall resource requirements and favors block ciphers that can share resources for encryption and decryption.

OCB does not provide incremental MAC and does not provide misuse resistance. It strictly requires the use of a new IV for every message!

While, to the best of our knowledge, no high-speed implementations of AES have been published, an analysis of OCB shows, that it should be able to reach the target throughput of 100 Gbit/s and that OCB could even be faster than the existing GCM implementation while requiring less resources. However, OCB requires encryption and decryption functionality of the underlying block cipher and has an increased latency every 64th message. Furthermore, the OCB decryption has a delay of the authentication tag calculation that requires to introduce caching memory. Please note, that OCB is patented. However, its patents are freely licensed over a large space: open-source software, non-military software, and OpenSSL [62].

10.3.3 Comparison and Choice of an Alternative Mode for Authenticated Encryption

Table 10.2 compares the features and performance of various modes for authenticated encryption such as: *OCB*, *GCM*, *CWC*.

The OCB mode of operation as described in Section 6.6 in general is able to meet the throughput requirements of the QCrypt project. However, its characteristics favor some block ciphers with certain attributes over others. The goal of this Section is to discuss the advantages and disadvantages that arise from the use of OCB with the block-ciphers AES, Serpent and Twofish.

Table 10.2: Overview of modes of operation.

Feature	**OCB**	**GCM**	**CWC**
Patented	yes[a]	no	no
Parallellizable	yes (E+A)	yes (E+A)	yes (E+A)
Provable secure	yes	yes	yes
Cipher text expansion	τ	τ	τ
Online	yes	yes	yes
Incremental MAC	no	yes	no
Error pass	no	no	no
Only encrypt engine	no	yes	yes
Associated data auth	yes	yes	yes
Authenticator length	$0 \dots n$	$0 \dots n$	$0 \dots n$
Error propagation	no	no	no
Cipher invocations (init)	1	1[b]	2
Cipher invocations (crypt)	$\lceil \lvert M \rvert / n \rceil +$ 1.016[c]	$\lceil \lvert M \rvert / n \rceil + 1$	$\lceil \lvert M \rvert / n \rceil + 2$
Nonce length [bit]	96 $(1 \dots 2^{128})$	96 $(1 \dots 2^{64})$	88

[a] The patents are freely licensed over a large space: open-source software, non-military software, and OpenSSL.
[b] Cipher invocations needed if a 96-bit nonce is used.
[c] Cipher invocations needed if a counter is used a a nonce.

The main reason that OCB favors some block ciphers is the fact that it needs an encryption and a decryption operation in the OCB-decryption routine. A detailed analysis shows that the OCB-decryption most of the time needs the block-cipher decryption routine. There is only three cases where the encryption is needed. One time after a key change in

the initialization phase to compute $E_K(0^{128})$ and one time for every new message in order to compute the initial value of Δ. Another encryption is needed every new message in the finalization step to compute the tag τ as described in Algorithm 6.6.1. In addition, the encryption engine is needed for authenticated-only data.

So it requires that the block cipher implements both the encryption and the decryption. Thus, block ciphers that allow sharing of hardware resources for encryption and decryption as AES and Twofish are favored when using OCB. Another idea is to instantiate multiple encryption as well as decryption engines of the block cipher. This could help to avoid pipeline stalls as instant switches between encryption and decryption may not be possible. Hence, block ciphers that consume comparable little area are favored. Again this is true for the AES cipher.

Taking all pros and cons into account **we chose to implement OCB as an alternative to GCM**. It, is well-analyzed and allows to achieve a throughput that even surpasses the throughput of GCM. While being well-suited for high-speed hardware implementations it has the disadvantage of requiring encryption and decryption functionality of the underlying block cipher which favors certain block ciphers. Furthermore, for decryption, the delay of the authentication tag calculation is higher compared to GCM. Also note, that OCB is patented. However, its patents are freely licensed over a large space: open-source software, non-military software, and OpenSSL.

Chapter 11

Serpent - Implementation of an Alternative Block Cipher

The aim of this section is to describe the practical implementation of the Serpent block cipher and to point out the design decisions made in order to suit the target system. In order to reach encryption throughputs exceeding 100 Gbit/s on today's commercial FPGA devices it is necessary to make use of multiple cipher instances. Additionally, pipelining has to be introduced to speed up the single instances. First, in Section 11.1 the structure of a single Serpent instance is described and then in Section 11.2 the overall implementation that consists of multiple Serpent instances and that is capable of encrypting at a throughput of over 100 Gbit/s is explained.

11.1 Single-Core Design

As the Serpent block-cipher consists out of 32 very similar rounds a natural approach to increase speed as well as throughput is to introduce 32 pipeline-stages—one after each round. Additionally, if required by the overall architecture, an additional input-register can be added. Due to the fact, that Serpent operates with 128-bit values this results in an overall register requirement of $32 \times 128 = 4{,}096$ bits if no input register is used and $33 \times 128 = 4{,}224$ bits if an input register is used just for the pipeline stages.

11.1.1 The Substitution Stage

As stated in Section 5.2.2 the *substitution stage* of Serpent makes use of S-boxes that perform a four-bit to four-bit mapping. On an FPGA there are basically two common ways to realize S-boxes. First, one could use the FPGAs internal RAM to implement the substitution or second, one could realize the substitution in pure combinatorial logic.

However, for the used device four-to-four bit substitutions are very inefficient when implemented in RAM. The Altera Stratix IV FPGA provides *block RAM (BRAM)* which as the name indicates can only be instantiated in blocks of relatively big size of either 9 kbit or 144 kbit. Additionally, as mentioned in Section 8.4, the FPGA contains 640-bit memory logic array blocks (MLABs) that can be configured as a simple 64×10-bit dual-port memory. The use of BRAM to implement the S-boxes is absolutely inefficient as only two 4-input S-Boxes can be realized in one 9 kbit or 144 kbit BRAM block even if it is used in dual-port

mode. Even when MLABs are used and configured as simple dual-port mode RAM only two 4-bit S-boxes can be implemented in one block which still results in a relatively bad degree of utilization. When taking into account that each round of Serpent—and thus each pipeline stage—requires 32 copies of the same S-box, one round requires 16 MLAB blocks of memory. As our Serpent implementation requires all 32 rounds to be accessible at the same time due to the insertion of the pipeline stages this results in an overall consumption of $512 = 16 \times 32$ MLABs without the S-Boxes needed for the key scheduler. This is an unacceptable overhead as this number of MLABs could hold 327,680 bits of data while in theory only $4 \times 32 \times 32 = 4{,}096$ bits are required to implement all S-Boxes needed in one instance Serpent.

On the contrary, the realization in combinatorial logic is highly efficient as the 4-bit S-boxes perfectly suit the structure of the Altera Stratix IV ALMs. Each ALM allows to realize two 4-input logic functions. So, one S-box can be realized using two ALMs and accordingly the 32 S-boxes required in one round of Serpent can be accomplished in 64 ALMs which equals 6.4 LABs. For the pipelined design this results in $64 \times 32 = 2{,}048$ ALMs that are needed for the implementation of the S-boxes which is a nice improvement compared to the block RAM based method and to a design based on MLAB memory, respectively. Consequently, we decided to realize the S-boxes as combinatorial functions.

11.1.2 The Linear Transformation

The *linear transformation* and its inverse used by Serpent is relatively cheap in hardware as it only uses fixed shifts, fixed circular shifts and XOR-operations, as shown in Figure 11.1. Fixed shifts and circular shifts are at no charge in hardware as they can be done using rewiring. As the linear transformation operates on 32-bit values and a maximum of two three-to-one XORs operate in series it has only little impact on the critical path and is very resource-friendly.

We made a modification to the original algorithm as described in Equations (5.3) and (5.4) by changing their general structure to the following form that makes the overall structure even more effective and that was also proposed in [64]:

$$\begin{aligned} \hat{B}_0 &\coloneqq IP(P) \oplus \hat{K}_0 \\ \hat{B}_{i+1} &\coloneqq R_i(\hat{B}_i) \\ C &\coloneqq FP(\hat{B}_{32}) \end{aligned} \tag{11.1}$$

where

$$\begin{aligned} R_i(X) &= L(\hat{S}_j(X)) \oplus \hat{K}_{i+1} & i &= 0, \ldots, 30 \\ R_i(X) &= \hat{S}_j(X) \oplus \hat{K}_{32} & i &= 31 \end{aligned} \tag{11.2}$$

This small modification transforms the 3-input linear transformation into a 4-input linear transformation while producing the same overall result. The 4-input linear transformation can be implemented much more efficiently in the Altera Stratix IV ALMs as they provide the possibility to implement two arbitrary 4-input logical functions. Thus, as each linear transformation contains four 32-bit four-to-one XORs it can be realized in $\frac{32 \times 4}{2} = 64$ ALMs which results in 1,984 ALMs for the pipelined version of Serpent.

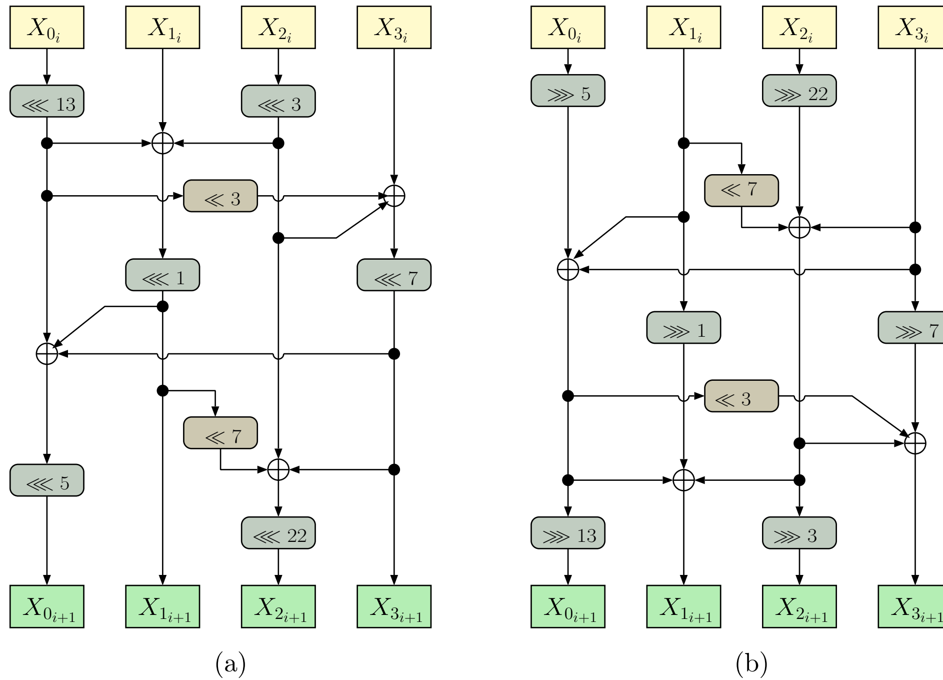


Figure 11.1: Overview of the linear transformation (a) and the inverse linear transformation (b) of Serpent.

11.1.3 The Key Scheduler

The most critical part of Serpent is its relatively complex *key scheduler*. One approach to compute the round keys is to compute them all in a single cycle. However, as shown in Figure 11.2 the calculation of the pre-keys which are necessary to compute a single round key involves a maximum of eight 128-bit 2-input XOR operations. So, when all 33 round keys are computed in a single cycle, this would form a path requiring $33 \times 8 = 264$ consecutive XORs and an additional run through an S-box. This would reduce the maximum clock frequency to a level unacceptable for a high-speed implementation.

One idea, to overcome this problem is to completely separate the key scheduler from the rest of the cipher and to use a dedicated, slower clock domain for the round-key calculation. Still, we decided not to follow this approach, as it brings various disadvantages but only few advantages. First asynchronous design is much more error prone than synchronous design and second, asynchronous design would make it necessary to introduce handshaking protocols between the different clock domains. This would introduce another level of complexity.

A nicer approach is to calculate one round-key per cycle on-the-fly. In any way, as pipeline stages were introduced, after a key change only one new round-key is needed per cycle. However, there is some aspect of Serpent that has to be considered when choosing this approach, as in the very last round of Serpent two round-keys are required during one cycle. To deal with this problem we decided to delay all the computed round-keys K_i but the very last round-key K_{32} by one cycle. Consequently, after the activation of a new user-key there is a delay of one cycle until new data can be encrypted using the actual key. In exchange for this one cycle delay, in the last round R_{31} both round-keys K_{31} and K_{32} are available. Please note, that when stating that the round-keys are calculated on-the-fly this is only

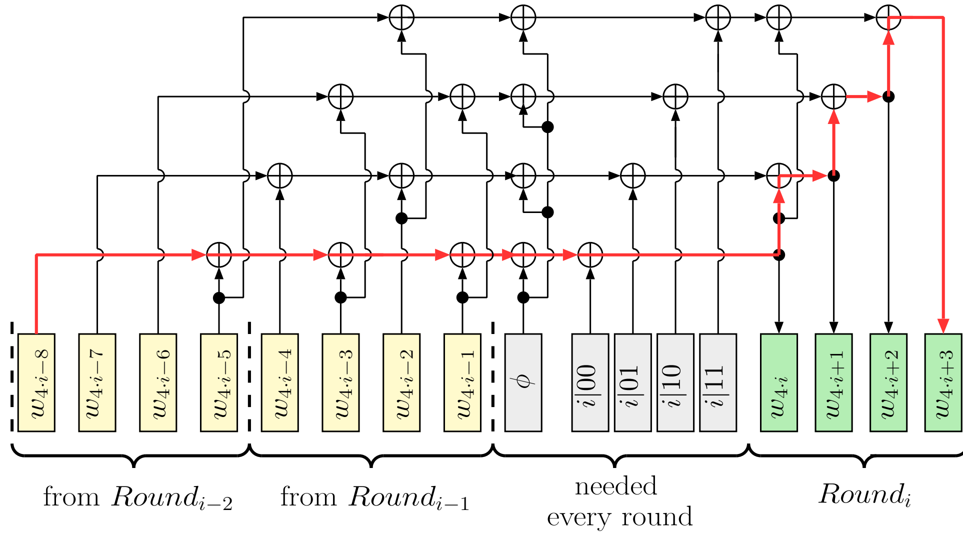


Figure 11.2: Overview of the critical path of the calculation of the pre-key values $w_{4i}, \ldots, w_{4i+3}$ of Serpent.

true for the very first cipher call after a key-change. In order to preserve the round-keys for following encryptions (or decryptions) they are stored in registers. This approach has the advantage of increased speed by reducing the critical path, as the round-key calculation gets separated from the actual cipher round. Furthermore, the additional cycle of delay that has to be introduced because of the calculation of round key K_{32} is eliminated for all cipher calls except the very first after a key-change. Still, in order to separate the round-key calculation from the cipher round and to preserve the round keys for future calculations it is necessary to introduce 33 128-bit registers.

As described in Equations (5.7) and (5.8) the calculation of a round key K_i includes the calculation of four 32-bit pre-keys $w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}$ which run through a substitution stage consisting of S-boxes. The calculation of the pre-key words $w_{4i}, \ldots, w_{4i+3}$, as shown in Equation (5.6) in turn requires the values of the pre-keys $w_{4i-8}, \ldots, w_{4i-1}$ to be available. Hence the design as described in Figure 11.3 includes two 128-bit registers that preserve the values of $w_{4i-8}, \ldots, w_{4i-1}$, the logic to calculate the pre-keys $w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}$, an S-Box block and 33 128-bit registers to store the round keys $\hat{K}_i$ where $i = 0, \ldots, 32$.

In case the 128-bit user key K has to be changed it is first padded to 256 bits by applying $\{1 \,\|\, 0^{127} \,\|\, K\}$ and by setting the signal *LoadUserKeyxSI* it is then loaded into the pre-key registers at the next clock cycle. Then the values of the pre-keys $w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}$ are calculated and passed over to eight *S-box blocks* for the S-boxes S_i where $i = 0, \ldots, 7$ where each of the S-box blocks contains 32 identical 4-bit input S-box copies. As described in Equation (5.7) the pre-keys required for round-key $\hat{K}_i$ have to be substituted using S-box $S_{(i+3 \mod 8)}$. Thus, a 128-bit 8:1 multiplexer selects the correct S-box block output and passes it over to the round-key registers. The correct round-key register is enabled using a 33-bit shift register that shifts the enable signal for the registers at each clock cycle.

In order to prepare for the calculation of the next round key the newly calculated values of $w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}$ are used as input for the pre-key register that holds the values of $w_{4i-1}, w_{4i-2}, w_{4i-3}, w_{4i-4}$ and the current values of this registers are used as input for the pre-key register that holds the values of $w_{4i-5}, w_{4i-6}, w_{4i-7}, w_{4i-8}$. This key-scheduler design offers the possibility to change the user key on the fly without loosing a single clock

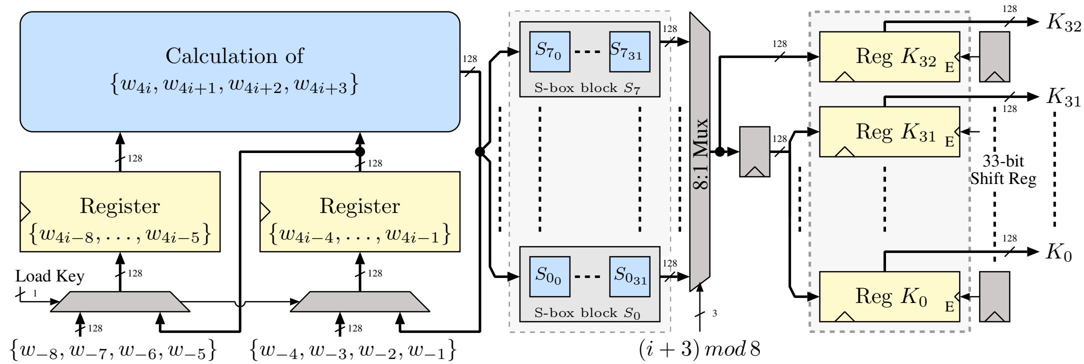


Figure 11.3: Overview of the key-scheduler design. (See Figure B.1 in Appendix B.1 for a more detailed figure.)

cycle. In order to do so, the secret user key has to be loaded three cycles before the first message should be encrypted using the new key. These three cycles result from one cycle required to load the user key to the w-registers, one cycle because of the delay register and an additional cycle to store the round key in the corresponding round-key register.

The critical path of the key scheduler includes the calculation of the pre-keys which requires eight consecutive XORs, a pass through the S-Boxes and finally a 8:1 multiplexer and thus also represents the critical path oft the whole Serpent design. Please see Figure B.1 in Appendix B.1 for a detailed overview of the key scheduler that allows to identify the critical path.

11.2 Multi-Core Design

As a single Serpent core is not sufficient to achieve a throughput of 100 Gbit/s multiple Serpent instances have to be used in parallel to accomplish that goal. Simulations of the single-core design suggested that three Serpent cores used in parallel could narrowly be enough to give the desired throughput. However, when considering that the maximum frequency is likely to be lower, when including the Serpent cores in the full QCrypt design compared to the single-core design we decided to use four serpent cores in parallel. As by design all four encryption cores use the same user key, only one key scheduler is required to generate the round keys. This results in a multi-core design as shown in Figure 11.4.

The four Serpent cores are used in parallel and use the same round keys that are provided by the common key scheduler. In order to reduce the overall power consumption the design provides enable signals for the pipeline stages of the cipher cores. These enable signals are generated using an external 1-bit signal that indicates that a valid plaintext block has to be processed. However, all four Serpent cores receive the same enable signals as we expect the outer logic to either deliver blocks of 512 bit or to discard invalid output blocks. Further, by shifting the enable signal through a 32-bit shift register it always coincides with the pipeline stage that has to be activated. In addition, the output of this shift register can be used as an outgoing signal indicating whether the output of the cipher cores is a valid cipher text or not.

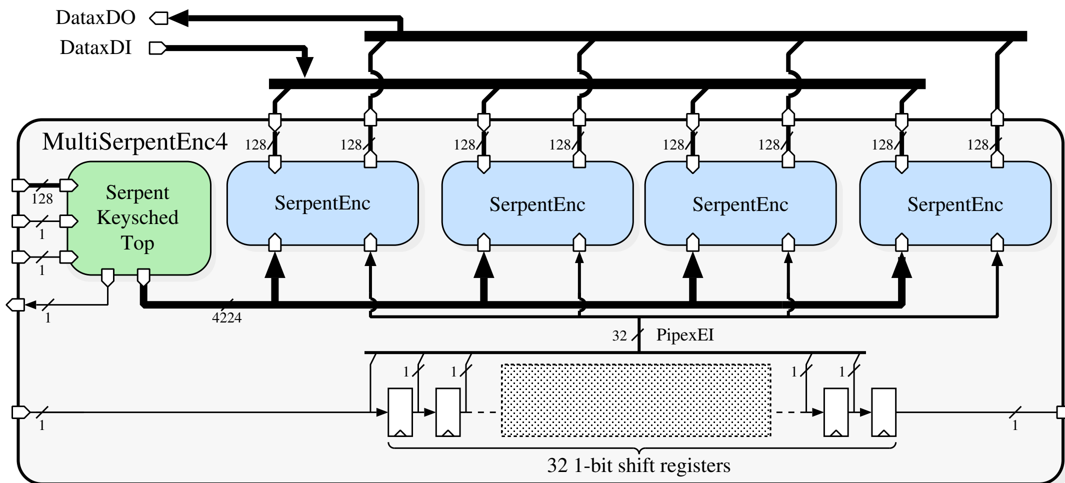


Figure 11.4: Overview of multi-core design with four instances of Serpent and a shared key-scheduler.

11.3 Summary

The Serpent block cipher was chosen and implemented as a promising alternative to the AES block cipher that is used in the existing AE engine. In order to achieve the required throughput of 100 Gbit/s the Serpent module uses four Serpent cipher cores in parallel that share one common key-scheduler. In addition, 32 pipeline stages were introduced to the single cipher cores. The key-scheduler architecture allows to compute the round-keys on the fly and features the pre-loading of a secret user key, in order to reduce the delay caused by key changes. Section 13 describes how the actual implementation was simulated and tested and Section 14.1 will give results on the throughput, maximum clock frequency and area requirements achieved for both, the single-core Serpent implementation as well as for the multi-core Serpent implementation.

Chapter 12

OCB - An Alternative for Authenticated Encryption

This section aims at describing the practical implementation of the Offset CodeBook mode of operation which we chose as an alternative to the existing authenticated-encryption engine, that is based on GCM. In addition, this section tries to point out the design decisions made in order to suit the target system. In order to reach encryption throughputs exceeding 100 Gbit/s on the target device it is necessary to make use of multiple cipher instances.

For the OCB architecture, we assume the following prerequisites:

- The size of the message block counter i is restricted to 7 bits, as a full Ethernet frame in IEEE 802.3ba has a maximum size of 1522 bytes. So 2^7 message blocks are sufficient to hold an entire ethernet frame.
- As the network interface ensures solely full message blocks, we do not handle short final message blocks separately.
- We assume that the handling of delayed authentication-tag calculations is handled by higher level units.

As OCB encryption and decryption differ in large parts, first, in Section 12.1 we describe the architecture of the OCB encryption core and second, in Section 12.2 we depict the architecture of the OCB decryption. Later, in Section 12.3 we describe the actual implementation of OCB with AES as the underlying block cipher. We refer to this composition as OCB-AES. Finally, we give an overview of the implementation of OCB with Serpent as the underlying block cipher, which we refer to as OCB-Serpent.

12.1 Encyption

Similar to the existing encryption engine as outlined in Chapter 9, multiple cipher cores have to be used in parallel to achieve extremely high throughputs of over 100 Gbit/s. Similar to GCM, OCB also allows two successive message blocks to be processed independently of each other. In order to be conform with the existing AE engine and to ease replacement, we took advantage of this fact and decided to use four block-cipher cores that share a common key-generation unit. The following description of the OCB encryption, will treat the cipher

cores as black boxes. Details, on the actual implementations with concrete ciphers are then given in Section 12.3 that describes the OCB-AES implementation and in Section 12.4 that introduces the OCB-Serpent implementation. Figure 12.1 illustrates the OCB architecture for authenticated encryption based on four block-cipher cores.

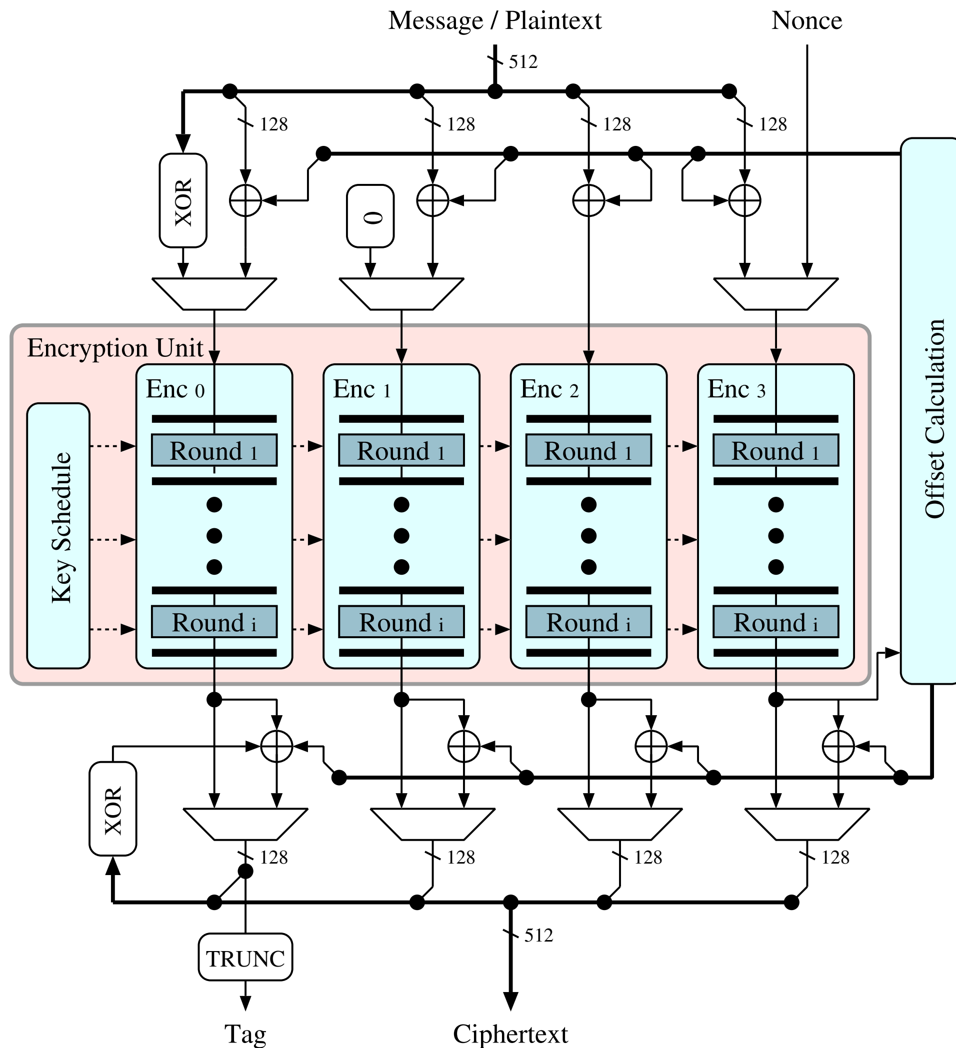


Figure 12.1: Simplified OCB-Encryption architecture.

OCB can, in general, be subdivided into three stages: *Initialization*, *encryption/authentication*, and *finalization*.

During the initialization phase, two potential pipeline stalls may occur if not handled properly. First, after each key change a cipher call $E_K(\varnothing)$ is required in order to be able to compute the table values $L[..]$ (cf. Algorithm 6.6.1). As, indicated in Figure 12.1, the control unit can apply the empty string $\varnothing$ to one cipher core using a multiplexer. Second, each new message needs a fresh nonce N, and thus a new offset value Δ. So, the initialization phase after a key-change causes a delay of $l+1$ clock cycles where l is the latency of the underlying block cipher.

The limitation of message lengths to a maximum of 2^7 blocks, facilitates the precomputation of the $L[..]$-values, as it limits the maximum number of trailing zeroes of the block

counter i to six. Thus, only $L_{\$}$, L_*, and $L_0 \dots L_6$ have to be precomputed, as shown in Figure 12.2. The, *double()* operation (cf. Equation (6.4)) only depends on operations that are cheap to implement in hardware, i.e., fixed shift- and conditional exclusive-or-operations with the constant `0x87`. In fact, when the result of $E_K(\varnothing)$ is available, all nine table values can be computed and stored in registers in a single clock cycle. So overall nine 128-bit register are needed to hold all $L[..]$-values.

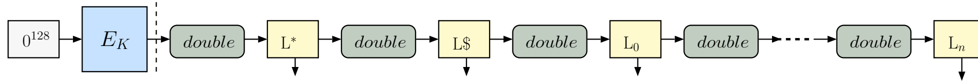


Figure 12.2: Calculation of the $L[..]$-values.

Since the calculation of the initial offset also requires a cipher call, this may result in another pipeline stall. As described in Algorithm 6.6.3, a nonce-dependent call to the encryption of the block cipher is required. The result of this operation, further-on called *Ktop*, then has to be shifted by a 6-bit nonce-dependent value *Bottom*. First, to be able to perform this shift-operation, *Bottom* has to be buffered until the result of $E_K(Top)$ is available. Second, the 6-bit variable shift is done using a 192-bit by 6-bit barrel shifter. Although, using a counter for the nonce N could avoid the resource-expensive barrel shifter, we decided to keep it in order to stay independent of the actual structure of the chosen nonce. **Note, that the use of a counter for the nonce is highly recommended.** It allows to minimize the block cipher calls, to compute the initial delta values, to only one block cipher call every 64th message. This comes from the fact that $Ktop = E_K(top)$ stays unchanged for 64 messages, when using a counter because $top = (1^{122} \| 0^6) \wedge N$. However, as bottom is the last six bits of a counter that increases by one per message, the nonce-dependent shift in Line 6 of Algorithm 6.6.3, is a fixed left-shift by one. In order to reduce the number of pipeline stalls to a minimum, we allow to precompute one initial offset value and to pass it over to the delta calculation when needed. By doing so, only one clock cycle is wasted. However, the logic when to start the precomputation is expected to be handled by the higher-level units.

When processing a block in authenticated-encryption mode, the message M_i is XORed with the current offset Δ_i, encrypted and finally XORed with Δ_i again. As pipeline stages were introduced into the block cipher, the Δ-values either have to be stored or recalculated. We chose the approach to recalculate the offset-values as it makes the implementation less dependent on the underlying block cipher and the number of pipeline stages it uses. Furthermore, the multi-core design is able to process four message blocks in parallel, consequently the offset-calculation units need to be capable of providing four offset values per cycle. In fact, the calculation of the four offset values is relatively cheap, as $ntz(i)$ is fixed for all $i \neq 4 * n + 3$, as shown in Figure 12.3. Thus only one $ntz(i)$-calculation unit is needed to be able to compute the Δ_{4*i+3}-values. All other Δ-values can be computed using XORs with fixed positions of $L[..]$. So, the computation of the Δ-values requires only four 128-bit XORs and has a critical path that is formed by four consecutive XORs.

When processing authenticated-only blocks, the message is XORed with the current offset Δ_j and XOR-ed with the intermediate authentication tag *auth*. Note, in Algorithm 6.6.4, that processing AAD blocks requires the, Δ-values to be computed slightly different compared to authenticated and encrypted blocks. Thus, the first offset-calculation unit that is followed by the encryption cores, additionally needs to be able to provide offset values for authenticated-only message blocks. A multiplexer is then used to select between

Figure 12.3: OCB calculation of the Δ-values for authenticated and encrypted message blocks.

the Δ-values for blocks in authenticated-encryption mode and AAD blocks.

In order to be able to compute the final authentication tag, the plaintext of all messages that are authenticated and encrypted are XORed in order to compute a checksum. The checksum is then XORed with $\Delta_{\$}$, encrypted and the result of the encryption is then XORed with the intermediate authentication tag *auth*. $E_K(checksum)$ can be computed right after the very last input message. So, for OCB encryption the delay of the authentication tag T is only one clock cycle.

12.2 Decryption

Authenticated decryption according to OCB is very similar to the encryption process. Exchanging the encryption operation E_K of the underlying cipher by the decryption operation D_K in line 8 and 13 of Algorithm 6.6.1, turns OCB into decryption mode. However, the encryption operations during the initialization phase, described in Algorithm 6.6.2 and Algorithm 6.6.3, as well as for the computation for the final authentication-tag calculation, remain. The requirement of block-cipher encryption and decryption is a drawback for hardware implementations. In order to be able to process authentication-only blocks with a throughput of 100 Gbit/s it would be necessary to provide four encryption cores. If the underlying block cipher is not able to share resources between encryption and decryption this would result in a large resource consumption overhead. However, pure authentication is not of primary importance as the system is used to authenticate and encrypt ethernet frames. Typically, as pointed out in Section 10.1, either the whole frame is encrypted and authenticated and embedded in a new frame, or only the header has to be authenticated only and the payload is authenticated and encrypted. The header of an IEEE 802.3 Ethernet frame is 14 bytes wide and thus fits into a single 128-bit block, which can then be authenticated. Thus, we decided to use four block-cipher decryption cores and one encryption core that is required during the initialization, for authentication-only message blocks and for the finalization. Still, all cipher cores share a common key-generation unit. This results in an architecture as shown in Figure 12.4.

A minor drawback of authenticated decryption according to OCB is the fact that a delay, depending on the number of pipeline stages p between the calculation of the plaintext and the calculation of the authentication tag, exists. This delay is caused, since the calculation of the authentication tag requires the encryption of the *Checksum* of the

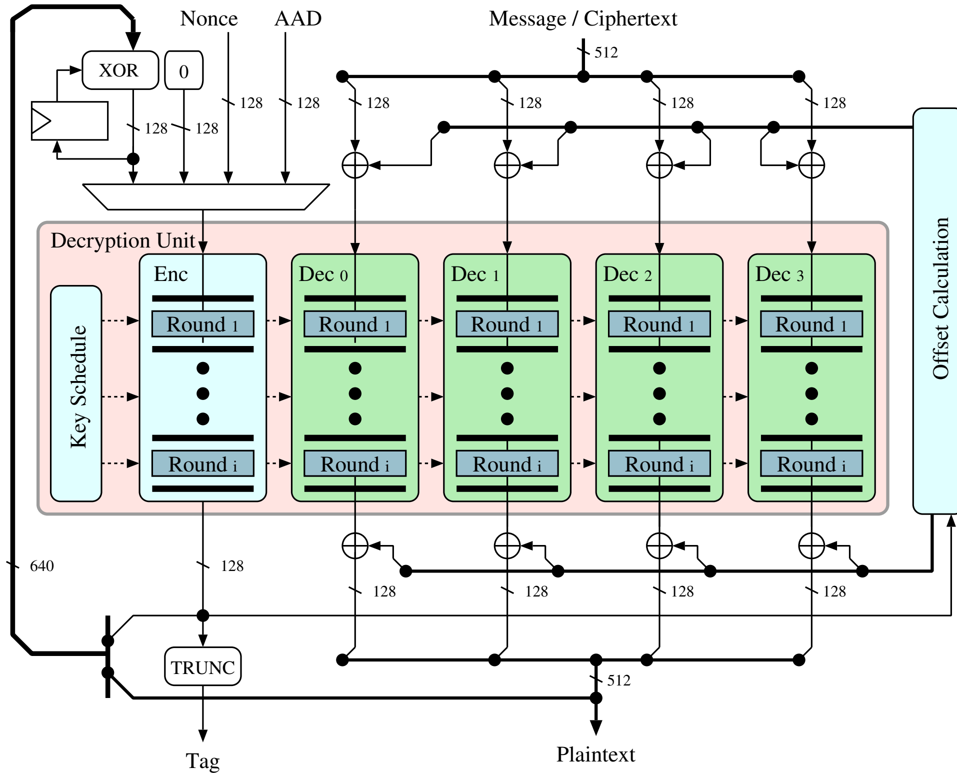


Figure 12.4: Simplified OCB-Decryption architecture.

plaintext. However, the $E_K(checksum)$ can only be computed after all message blocks that are authenticated and encrypted have been decrypted and XOR-ed to give the overall *Checksum*. If not treated specially, this would lead to l pipeline stalls, where l is the latency of the underlying block cipher. Thus, in order to reduce the number of pipeline stalls, we allow to start the decryption of a new message M_{i+1} before the authentication tag T_i of message M_i has been calculated. As soon as the *checksum* is ready, the authentication tag is computed. Still, in order to be able to verify the authentication tag of a message, the entire plaintext message has to be buffered which results in additional memory requirements of 1522 bytes for one message as IEEE 302.3 ethernet frames allow a maximum size of 1522 bytes including payload and header. As it is allowed to start the decryption of M_{i+1} before the authentication tag T_i has been calculated, it is necessary to also store the plaintext blocks of M_{i+1} until the new message tag T_{i+1} has been computed. The number of messages allowed to be applied before T_i is computed, and consequently also the number of plaintext messages that need to be stored depends on the latency of the underlying block cipher and is treated in the sections that treat the implementation of OCB with the block cipher AES and the block cipher Serpent.

12.3 OCB-AES

This section describes the implementation of OCB with AES as the underlying block cipher that we refer to as OCB-AES. The overall structure, of OCB-AES remains as outlined in Figures 12.1 and 12.4. However, due to the characteristics of the high-speed AES implementation as described in Section 9.1 some parts of OCB have been optimized, in

order to reduce latency and pipeline stalls.

The existing AES implementation has a latency of 21 clock cycles. So, the initialization phase after a key change needs 21 cycles for the calculation of $E(\varnothing)$ plus one cycle to compute the initial Δ-value. Another cycle is needed to perform the precalculation of the values of Δ_1, Δ_2 and Δ_3 that are required as four cipher cores are used in parallel. So, after 23 clock cycles OCB-AES is ready to authenticate and encrypt messages.

As described in the previous section, it is possible to start the decryption of a message M_{i+1} before the authentication tag T_i for message M_i has been calculated. As four cipher cores with a block size of 128 bits are used in parallel, 512 bits or 64 bytes can be processed per pipeline stage and clock cycle, respectively. So when all pipeline stages are filled, $64 \times 21 = 1{,}344$ bytes are processed in the pipeline. As the majority of ethernet frames is expected to be of the maximum ethernet frame size of 1,522 bytes, we decided to allow the message M_{i+1} to be applied before T_i is computed. This way it is possible to effectively reduce the number of pipeline stalls to zero if full-length frames are processed (except for every 64th message which requires the recomputation of the initial Δ-value). Still, for shorter messages pipeline stalls may occur. Allowing one message to be applied, requires to store the last Δ-value and the intermediate authentication tag *Auth* for message M_i. In addition it is necessary to provide memory for two entire frames of 1,522 bytes size. On the one hand, M_i has to be stored until the tag T_i has been computed and verified, and on the other hand, the blocks of plaintext that are computed until the final *Checksum* of message M_i is ready have to be preserved. The necessary memory can be realized using two BRAM M144K blocks[1] of a total of 64 M144K blocks available on the Altera Stratix IV EP4S100G5 FPGA.

Apart from the mentioned adaptions, only the control logic has to be adapted to fit the latency of the AES cores and the number of cycles required for the setup of the ciphers key-generation unit.

12.4 OCB-Serpent

This section outlines the implementation of OCB with Serpent as the underlying block cipher that we refer to as OCB-Serpent. Again, the overall structure of OCB-Serpent remains as shown in Figures 12.1 and 12.4.

Basically, the same adaptions and optimizations as for OCB-AES have also been realized for OCB-Serpent. However, due to the larger latency of 33 cycles of the Serpent implementation described in Chapter 11, compared to the AES implementation, these have to be tuned at some points. As, Serpent has a latency of 33 cycles the number of cycles required for the initialization phase after a key change is $33 + 1 + 1 = 35$ clock cycles. This is 43% higher compared to the initialization delay of OCB-AES. In addition, the measures that allow to start the decryption of a message M_{i+1} before the authentication tag T_i for message M_i has been calculated have to be adjusted. When all 33 pipeline stages of Serpent are filled, $64 \times 33 = 2012$ bytes are processed in the pipeline. This is more than the maximal length of an ethernet frame. So, in order to reduce the number of pipeline stalls to zero (except for every 64th message) it has to be allowed to start the decryption of two new messages M_{i+1} and M_{i+2} before the computation of the authentication tag T_i is started.

[1] M144K blocks are embedded memory blocks on Stratix IV FPGAs that provide 144 kbit of memory, which can be configured as single-port RAM, dual-port RAM, shift registers or ROM with variable word lengths. See [9] for more information.

As a consequence, two Δ-values and two intermediate authentication tags $Auth$ have to be stored. Furthermore, it is necessary to be able to store three ethernet frames of 1522 bytes size. This requires 3 M144K blocks of 64 M144K block available on the target FPGA.

Again, apart from the mentioned adaptions, only the control logic has to be adapted to fit the latency of the Serpent cores and the number of cycles required for the setup of the cipher's key-generation unit.

Chapter 13

Simulation and Verification

Today's hardware systems implemented on FPGAs, such as the one described in this thesis, are very complex and consist of many modules interacting with each other. For such systems it is a difficult task to ensure the correct behavior for all possible input sequences and operating conditions.

In order to guarantee correct functionality, all submodules, modules and finally the entire system have to be simulated and verified. For the simulation a dedicated software is used that models the design at different levels of abstraction, starting from the behavioral descriptions of the system and ending with a model derived after RTL synthesis and place-and-route. Dedicated hardware modules that allow to apply bits at defined positions inside the core and to read out the values of any internal register inside the core were added on the FPGA for verification and debugging purposes.

Within this thesis, simulation and verification was performed at the following levels of abstraction:

1. Behavioral Model
2. After place-and-route
3. Bit-file on FPGA

For levels 1 and 2 file based simulation as described in Section 13.1 was used for the verification of the submodules and the whole cores. Finally, the bit files of the different block-cipher cores and the different authenticated-encryption mode and block-cipher combinations were verified using the FPGA testbench model explained in Section 13.2.

13.1 File-based Simulation

For each of the cores developed within the scope of this theses i.e. the Serpent core, the GCM-AES core, the GCM-Serpent core, the OCB-AES core and the OCB-Serpent core a golden model that generates the appropriate stimuli and response vectors for the cores as well as their submodules was implemented. For Serpent we developed a bit-accurate model of the architecture in C++ that allows co-simulation of the VHDL model. The golden model performs a self-test using test vectors provided in the AES competition submission package provided in [104]. In order to create test vectors for the authenticated encryption

engines we implemented golden models using the Crypto++® Library version 5.6.2 [30]. As this library does not feature the OCB mode, we extended the library with an OCB implementation. Again, all golden models perform a self-test using the test vectors provided in the algorithm specifications[1] A testbench, as described in Figure 13.1, which is entirely implemented in VHDL, then reads out the appropriate vectors created by the golden models and applies them to the device under test (DUT) after a specified input delay. An advantage of implementing the testbench in pure VHDL compared to other testbench approaches (e.g. using Tcl[2]) is that it is known to be much faster when simulating thousands of test runs.

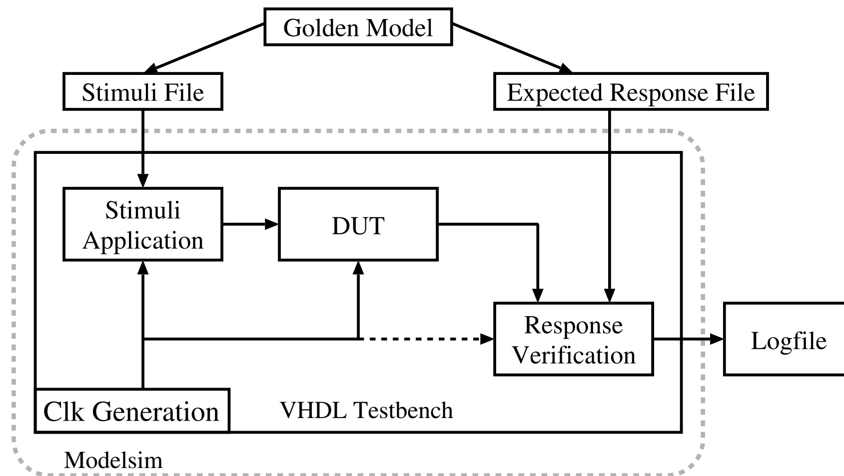


Figure 13.1: Block diagram of the file based simulation and verification.

For smaller, less complex modules the golden model generates cycle accurate stimuli and expected response files that also include intermediate values. The testbench reads and applies the provided stimuli and compares the output of the DUT with the expected responses every clock cycle. The result of the comparison as well as some debugging information is then written to a log file.

For complex modules and more sophisticated tests that aim to cover as many of the internal states of the DUT as possible this approach is not practicable. If stimuli and expected responses would be provided for every clock cycle, the test files would become too large and contain a lot of unnecessary data. Thus, for complex modules, the testbench is designed to react on signals provided by the DUT. Stimuli are only applied to the DUT when it signals to be ready and expected responses are checked whenever the DUT indicates a valid output. Again, debugging information is written to a log file. In order to provide a good test coverage the golden model allows to generate stimuli and expected response pairs for test cases defined in the descriptions of the implemented algorithms (whenever such pairs are provided), for predefined debugging-friendly input and for random input stimuli.

To ensure the correct functionality after RTL synthesis and place-and-route the model provided by the FPGA-vendors synthesis software performing these steps was again included in a testbench implemented in VHDL and checked against the stimuli and expected-response pairs provided by the golden model.

[1] Functional correctness of the golden models of GCM and OCB was verified with AES as the underlying block cipher, as both algorithm specifications only provide test vectors for this cipher.

[2] Tcl is a scripting language and many digital logic simulator vendors use it to interface with the HDL implementations.

13.2 FPGA Testbench Design

After successful simulation, two different testbench versions that are suited to run on the actual QCrypt fast-encryptor FPGA board were implemented for each of the main cores, namely the GCM-AES, the GCM-Serpent, the OCB-AES and the OCB-Serpent. The first version tests the authenticated-encryption engines as standalone modules while the second version checks the AE engines in combination with the network interface.

The first testbench version, as shown in Figure 13.2, uses a small set of stimuli vectors as well as the corresponding set of expected responses that are loaded into block RAMs on the FPGA. A simulation control unit takes these vectors, applies them to the DUT and compares the responses created by the core with the provided expected responses. In order to give some status information on the current test run error codes are provided through the encryptor board's LEDs. In addition, to ease debugging and to be able to inspect buses and on chip data, logic analyzers generated by SignalTap® II Logic Analyzer were used. The SignalTap® II Logic Analyzer is a tool provided by Altera that allows to capture and debug the values of internal signals via JTAG[3], without using extra I/O pins, while the design is able to run at full speed on the FPGA device. In order to be able to capture a signal on the FPGA it has to be selected before place-and-route. The logic to debug a signal is then automatically included into the design.

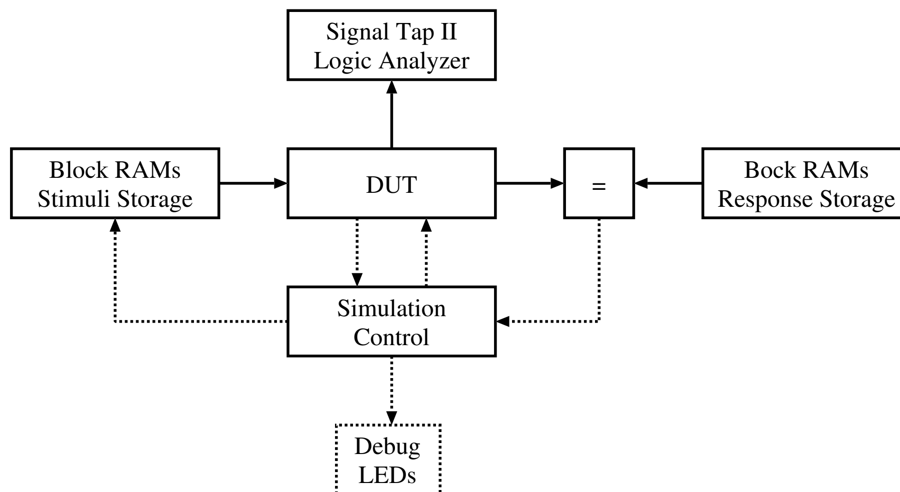


Figure 13.2: Block diagram of the standalone AE-engine FPGA testbench design.

Furthermore, in order to perform a run-time test under realistic conditions and to verify the interaction of the AE-engine with the networking components a test system, as shown in Figure 13.3 was developed and executed on the encryptor board. It uses the existing network interface with the sender part directly connected to the receiver part and includes one AE engine configured as encryption unit for the sender part and one AE engine configured as decryption unit for the receiver part. The simulation control generates test data using a 128-bit counter which is then applied to the AE engine and encrypted and/or authenticated using pseudo random control sequences. The output of the AE engine is then passed over to the sender's network interface and transmitted to the receiver's network interface which hands it over to the senders decryption unit. After decrypting

[3] Joint Test Action Group (JTAG) a commonly used name for the IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture. It is commonly used for IC debug ports.

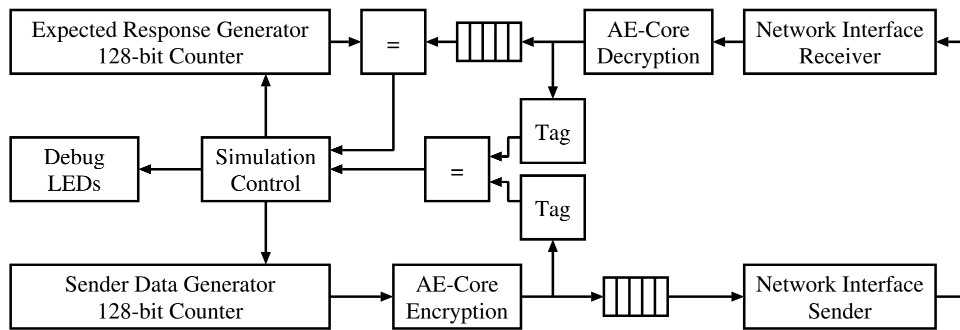


Figure 13.3: Block diagram of the FPGA run-time testbench design and its interaction with the networking components.

the incoming message the resulting plaintext is compared to the message created expected response generator and the resulting massage tag is compared to the sender's message tag. The outcome of this comparisons is then encoded and visualized using debug LEDs on the encryptor board. Again, we used the SignalTap® II Logic Analyzer for debugging and for inspecting internal signals on the FPGA.

Chapter 14

Results

Throughout this chapter, we present the implementation results of the designs, which figured out to be valid alternatives to the existing authenticated-encryption engine which is based on GCM-AES. The results are given in terms of throughput, resource consumption and efficiency on the Altera Stratix IV EP4S100G5 FPGA. First, Section 14.1, presents the results for the standalone Serpent block cipher, which was chosen as an alternative to the AES cipher used in the existing AE implementation. Second, in Section 14.2, we give the results for different authenticated encryption architectures we implemented. We present the results for GCM-AES, which represents the existing authenticated-encryption engine, in order to be able to compare it with the novel AE implementations. The actual implementation of GCM-AES is described in detail in Section 9. Next, we describe the results for GCM-Serpent which uses Serpent as an underlying block cipher. Furthermore, we give the results for OCB, which was chosen and implemented as an alternative mode for authenticated encryption as outlined in Chapter 12. These results include two versions of OCB that differ in the underlying block cipher. The first version is OCB-AES, which uses AES as the underlying block cipher, and which is described in detail in Section 12.3. The second version is OCB-Serpent, whose architecture is outlined in Section 12.4 and which applies Serpent as the underlying block cipher. All results presented in this chapter were achieved using the Altera Quartus II version 11.0 with speed-optimized settings for synthesis targeting the Altera Stratix IV EP4S100G5 FPGA. Functional correctness was verified using Modelsim 6.6e simulator.

14.1 Block Ciphers

This section presents the results for the existing AES cipher and the novel implementation of the Serpent cipher targeting the Altera Stratix IV EP4S100G5 FPGA. Both ciphers are analyzed and compared in terms of throughput, resource consumption, latency and the number of cycles that are required for initialization after a key change. However, the discussion of the results puts a strong focus on the Serpent cipher as it is one of the contributions of this work. Subsection 14.1.1 gives results for single-core implementations and treats the standalone cipher cores as well as the key generation modules. The achieved results for the multi-core architectures, which are required to achieve the target throughput of 100 Gbit/s are given in Subsection 14.1.2.

14.1.1 Single Core

Table 14.1 lists the results for single core implementations of the AES cipher and the Serpent cipher. All values presented are valid for plain instances of the components listed. There is no additional interfaces, buffers or glue logic included. The results include separate values for the standalone ciphers without key generation and standalone key schedulers, as the multi-core design that is required by the authenticated-encryption engines is composed out of a variable number of cipher cores that all share one common key scheduler.

Table 14.1: Results for single cores of AES and Serpent based on the Altera Stratix IV (EP4S100G5F45) platform.

	AES		**Serpent**	
	key scheduler	*cipher*	*key scheduler*	*cipher*
Number of ALMs	1,303	1,505	4,690	6,947
M9K BRAM Blocks	2	80	0	0
	Overall		*Overall*	
Number of ALMs	2,608		10,837	
M9K BRAM Blocks	82		0	
Max. Frequency [MHz]	252		284	
Max. Throughput [Gbit/s]	32.3		36.3	
Latency [cycles]	21		33	
Key Generation Enc/Dec [cycles]	2/20[a]		0/32[ab]	

[a] Encryption keys can be computed on-the-fly. One round key is updated per cycle. For decryption the last round key generated is required in the first round.
[b] If the secret key is applied one cycle in advance as supported by the Serpent architecture.

As expected, the standalone Serpent cipher core, without key scheduler, consumes a higher number of ALMs compared to AES. A standalone Serpent core consumes 6,947 ALMs while the AES standalone core requires only 1,505 ALMs. This is due to two main reasons. First, while Serpent's round function is relatively simple, Serpent has a higher number of rounds and consequently a higher number of round functions and pipeline stages have to be instantiated. As the Serpent design contains 33 pipeline stages, these require $33 \times 128\,\text{bits} = 4{,}224\,\text{bits}$ of registers, which can be provided by 2,112 ALMs. The, linear transformation contains four 32-bit four-to-one XORs and can be realized in $\frac{32 \times 4}{2} = 64$ ALMs which results in 1,984 ALMs for the pipelined version of Serpent. Second, the Serpent implementation uses a LUT-based approach to realize the S-Boxes, while AES utilizes BRAM for the S-Boxes. As discussed in Section 11.1.1, 2,048 ALMs are required to implement the S-Boxes for all 32 rounds of the Serpent cipher. Note, that the stated number of ALMs required for the building blocks of Serpent cannot simply be summed up to get the overall resource consumption. Each Stratix IV ALM provides multiple resources such as two LUTs, two Registers, two adders and some multiplexers that can be shared among the building blocks. Thus, the effective number of ALMs required is lower than the simple sum of ALMs required for the subcomponents. As such, the values given in Table 14.1 can be seen as upper bounds.

The Serpent key scheduler is comparably complex and requires 4,690 ALMs which is a significantly higher resource requirement compared to the AES key generation which needs

1,303 ALMs. First, the key scheduler of Serpent requires eight different sets of S-Boxes even if only one round key is computed per cycle. This consumes 512 ALMs as the S-Boxes are implemented using LUTs. In addition, to select the correct S-Box output, a 128-bit 8:1-multiplexer is necessary. Second, two 128-bit registers that hold the intermediate w-values, one 128-bit register used to delay the round keys by one cycle, and 33 128-bit registers that store the round keys are needed in the design. This results in 36×128 bits = 4,608 bits of registers required overall. Furthermore, the key scheduler requires 20 32-bit 2:1 XORs with a maximum logic depth of 8 consecutive XORs.

Due to its complexity, the key scheduler also forms the critical path of the entire Serpent architecture. The critical path is formed by eight levels of XORs, a pass through the S-Box, an 8:1 multiplexer[1] and a register (see Figure B.1 in Appendix B.1 for a detailed overview of the key scheduler). This results in a maximum frequency of 284 MHz for the overall Serpent core, including the cipher and the key scheduler. This maximum frequency leads to a maximum throughput of 36 Gbit/s which is an increase of 12.7 % over the existing AES design which reaches 32 Gbit/s at a maximum frequency of 252 MHz.

As the Serpent architecture applies one pipeline stage for every round of Serpent and adds an additional input register, this gives a latency of 33 clock cycles. Due to the lower number of rounds, the AES design comes with a latency of only 21 cycles. Consequently, pipeline stalls have a bigger impact with Serpent than with AES. This is especially true if the result of the encryption $E_K(I_i)$ of input I_i is needed to be able to compute $E_K(I_{i+1})$ as occurring at some points in the OCB algorithm. In this case, for Serpent, the pipeline is stalled for 33 cycles while $E_K(I_i)$ is computed and then 33 cycles are needed until $E_K(I_{i+1})$ is computed. So the overall delay to compute $E_K(I_{i+1})$ would be 66 cycles after applying I_i for Serpent while it would only be 42 cycles for AES. When taking the higher maximum frequency of Serpent into account this gives a delay of 232 ns for Serpent and a delay of 167 ns for AES to compute $E_K(I_{i+1})$ if the input I_{i+1} is dependent on $E_K(I_i)$.

In order to minimize the cost of key changes, the designs of AES and Serpent allow to compute the round keys on-the-fly. So each clock cycle, the keys for one more round are updated, while the rest of the old round keys is preserved for pending calculations in the pipeline. For encryption it is thus possible to perform a key change with loosing a minimal number of clock cycles. After applying a new key to the AES key generation module it is necessary to wait for two more cycles until new messages can be applied. The Serpent architecture even allows key changes without loosing any clock cycle if the new key is applied three cycles before the first message has to be encrypted using this new key. Still, AES and Serpent both require that the round keys are applied in reverse order for decryption. However, as the computation of the last round key is dependent on all previous round keys, these have to be computed beforehand. This means, that 20 cycles are required for AES and 32 cycles are needed for Serpent until the first message can be decrypted using the new key.

14.1.2 Multi Core

As a single cipher core is not sufficient to reach the target throughput of 100 Gbit/s, multiple instances have to be used in parallel. By design all cipher cores use the same user key and so only one key scheduler is required to generate the round keys. This section presents the results for the multi-core designs that are used by the authenticated encryption engine.

[1] An 8:1 multiplexer has to be assembled from multiple multiplexers which increases the logic depth.

Table 14.2 lists the results for the multi-core encryption and decryption architectures for AES and Serpent. All values are valid for the standalone modules without any additional input or output buffers, interfaces or glue-logic. The multi-core design for encryption uses four cipher blocks in parallel that share a common key scheduler. As required by the OCB-decryption architecture which is described in Section 12.2, the multi-core decryption architecture includes four decryption cores and one encryption core that all share a common key scheduler.

Table 14.2: Multi-core encryption and decryption results for AES and Serpent based on the Altera Stratix IV (EP4S100G5F45) platform.

Cipher	**Mode**	**Cipher Cores**	**Area**		$\mathbf{f_{max}}$	**Throughput**
			[ALMs]	[M9K Bl.]	[MHz]	[Gbit/s]
AES multi	E	4	6,916	322	252	129
Serpent multi	E	4	25,825	0	275	140
AES multi	D	5	8,793	402	247	126
Serpent multi	D	5	31,478	0	273	140

The results for the multi-core architectures confirm the findings of the single-core variants. The multi-core Serpent encryption design requires 25,825 ALMs which is a significantly higher resource consumption compared to the multi-core AES design which requires 6,916 ALMs. However, it has to be considered that the AES architecture requires 322 M9K BRAM blocks of 1,280 M9K blocks available on the target FPGA, while Serpent does not use any BRAM at all. As the network interface of the fast-encryptor system heavily relies on the use of BRAM, routing overhead increases for AES. However, as the Altera Stratix IV (EP4S100G5F45) FPGA features 212,480 ALMs, the multi-core Serpent encryption architecture consumes only around 15 % of the ALMs available. So, the resource consumption of Serpent is at an acceptable level.

The decryption architectures, that are needed for OCB decryption use a common key scheduler and four decryption cores in parallel and also include one cipher core for encryption. Consequently, the decryption designs consume more resources compared to the encryption designs. The multi-core AES decryption architecture consumes 8,793 ALMs and 402 M9K BRAM blocks while the multi-core Serpent decryption design requires 31,478 ALMs and no BRAM.

Still, the maximum frequency of the multi-core variants of AES and Serpent is lower compared to the single-core variants due to the increased routing overhead. Overall, this results in a maximum frequency of 252 MHz for the multi-core AES encryption module which translates to a throughput of 129 Gbit/s. The multi-core AES decryption module is slower because the *InvMixColumns* operation is more complex than the *MixColumns* operation. In addition, the increased routing overhead due to the additional AES core also decreases the maximum frequency. AES-decryption reaches a maximum frequency of 247 MHz which gives a throughput of 126 Gbit/s. The multi-core Serpent encryption module reaches a throughput of 140 Gbit/s at a maximum frequency of 275 MHz. This is an increase of 8.3 % compared to AES. A throughput of 140 Gbit/s at a maximum frequency of 247 MHz can be achieved for the multi-core Serpent decryption design. Because Serpent's encryption and decryption operations are of similar complexity, also the multi-core encryption and decryption reach a relatively similar maximum frequency. In fact, the decryption is only

slower because of the increased routing overhead due to higher resource consumption compared to the multi-core Serpent encryption.

All four multi-core architectures are able to achieve the target throughput of 100 Gbit/s using four cipher cores in parallel for encryption and a total of five for decryption as it is required by the OCB architecture. The Serpent shows a higher throughput compared to AES, but also requires significantly more ALMs. In fact, Serpent encryption could reach the target throughput with only three cipher cores in parallel when used as a standalone design. However, when included in the overall fast-encryptor system this may not be true, as large parts of the resources available are then consumed and the increased routing overhead lowers the maximum frequency. In addition, the chosen architecture complies with the existing AES architecture. This is necessary, as the network interface is designed to deliver four 128-bit strings of data in parallel. Changing this to only three message blocks would require extensive changes in the architecture of the network interface.

14.2 Modes for Authenticated Encryption

This section gives the results for the authenticated-encryption architectures additionally implemented. These are GCM-Serpent, OCB-AES, and OCB-Serpent. We compare the results with the existing authenticated-encryption engine, which is based on GCM-AES. Furthermore, we closely analyze the throughput of the different AE implementations that can be achieved under real life conditions on the target system. In addition, we present a comparison with related work.

Table 14.3 presents the results for all authenticated-encryption architectures that were implemented. In addition, the first two lines contain results for the standalone multi-core architectures of AES and Serpent for comparison and in order to elucidate the overhead caused by the authenticated encryption. All throughputs given in the table are valid under the assumption that the pipeline is completely filled at every cycle.

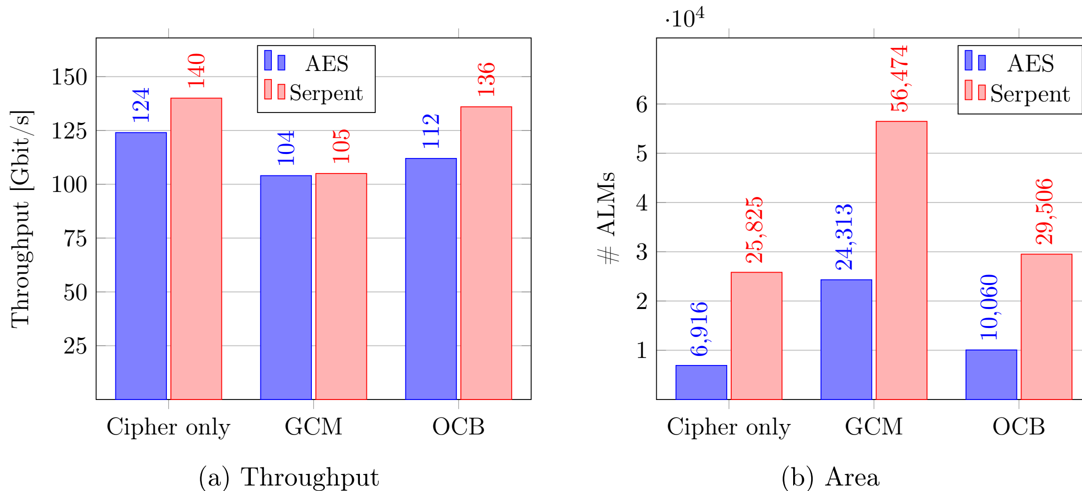
Table 14.3: Encryption-only and authenticated-encryption results based on the Altera Stratix IV (EP4S100G5F45) platform.

AE Mode	**Mode**	**Cipher Cores**	**Area**			**f_{max}**	**Throughput**	
			[ALMs]	[M9K Bl.]	[M144K Bl.]	[MHz]	[Gbit/s]	[%]
AES only	E	4	6,916	322	0	252	124	118
Serpent only	E	4	25,825	0	0	275	140	133
GCM-AES	E/D	4	24,313	322	0	206	105	100
GCM-Serpent	E/D	4	56,474	0	0	203	104	99
OCB-AES	E	4	10,060	322	0	220	112	107
OCB-AES	D	5	11,614	402	2	219	112	107
OCB-Serpent	E	4	29,506	0	0	267	136	127
OCB-Serpent	D	5	33,891	0	3	265	136	127

The existing GCM-AES architecture is the basis for the performance comparison. As GCM encryption and decryption have the exactly same structure, results for both modes are equal. GCM-AES consumes 24,313 ALMs and 322 M9K BRAM blocks for encryption and decryption respectively. It achieves a throughput of 105 Gbit/s at a maximum frequency of 206 MHz. This is significantly lower than the standalone AES which reaches a throughput of 118 Gbit/s. The decrease in throughput and maximum frequency is caused by the GHASH

core, which is the most complex part of GCM and which forms the critical path for the entire GCM architecture. This also affects the GCM-Serpent architecture. While the standalone multi-core Serpent design reaches a maximum throughput of 140 Gbit/s the GCM-Serpent design only reaches a throughput of 104 Gbit/s at a maximum frequency of 203 MHz while consuming 56,474 ALMs. So the throughput of GCM-Serpent is even lower than the throughput of GCM-AES. This can be explained by the higher resource consumption of GCM-Serpent. In consequence, the routing overhead is increased and the maximum frequency of the GHASH core is decreased. For the OCB architectures, results for encryption and decryption have to be treated separately as their design differs. OCB encryption uses four cipher cores in parallel. For the OCB-AES architecture this results in resource consumption of 10,060 ALMs and a throughput of 112 Gbit/s at a maximum frequency of 220 MHz. Compared to GCM-AES this is an increase in throughput of 7 % while only 41 % of the resources are needed. With a throughput of 136 Gbit/s at a frequency of 267 MHz, which represents and increase of 27 % compared to GCM-AES, OCB-Serpent reaches the highest throughput of all AE engines realized. In addition the design requires 29,506 ALMs which is only an increase of 21 % compared to GCM-AES. As described in Section 12.2 the OCB-decryption architecture uses four cipher cores in parallel for decryption and an additional cipher core that is in encryption mode and that is required during the initialization and the finalization phase. Again, all cipher cores share a common key scheduler. Using this design the OCB-AES decryption needs 11,614 ALMs and 2 M144K BRAM blocks and reaches a maximum frequency of 219 MHz which results in a throughput of 112 Gbit/s. The OCB-Serpent decryption uses 33,891 ALMs and 3 M144K BRAM blocks and reaches a throughput of 136 Gbit/s at a maximum frequency of 265 MHz. So, while OCB decryption achieves throughputs that correspond closely to those of the OCB encryption, the resource consumption increases due to the additional cipher core.

As illustrated in Figure 14.1 OCB exceeds GCM in terms of throughput. Both, OCB-AES and OCB-Serpent are significantly faster than GCM. OCB authenticated encryption has a relatively low overhead compared to the standalone cipher architectures.

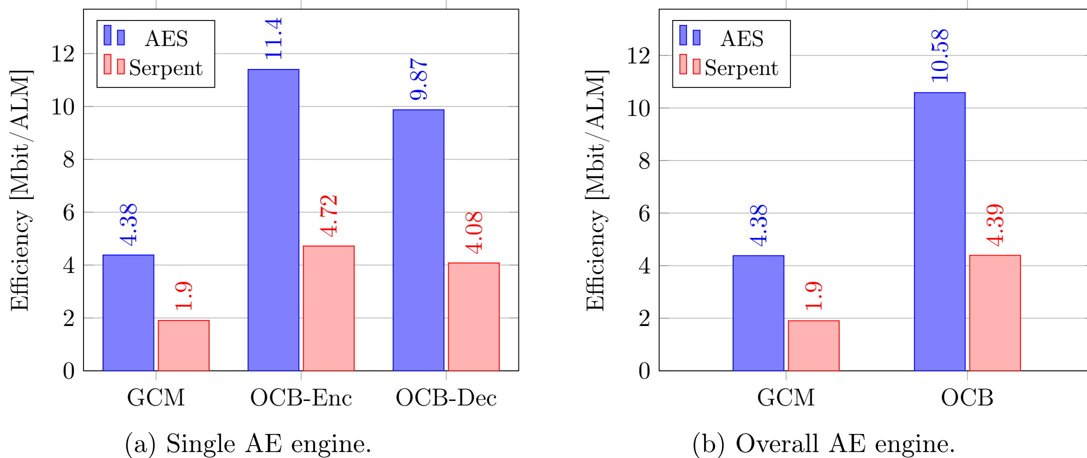


(a) Throughput (b) Area

Figure 14.1: Results for the implemented combinations of authentication-encryption modes and block ciphers. The results of the standalone block-cipher cores are included for comparison. Figure 14.1a shows the achieved throughput and Figure 14.1b displays the number of used ALMs. Note, that the graphic does not incorporate used BRAM blocks.

In terms of throughput OCB-AES has an overhead of 10 % compared to the standalone AES and OCB-Serpent has an overhead of only 2.2 % compared to the standalone multi-core Serpent design. OCB also is superior in terms of resource consumption. OCB-AES only needs 41 % of the ALMs required by GCM-AES and OCB-Serpent consumes only 52 % of the ALMs needed by GCM-Serpent. Note, that the area comparison between the AE architectures based on AES and those based on Serpent is not entirely fair, as AES uses M9K BRAM blocks, while Serpent actually does not. However, to the best of our knowledge there exists no formula to convert BRAM to an equivalent in ALMs. If the S-Boxes for AES were implemented using a LUT-based approach, the difference in consumption of ALMs would be far smaller.

The advantage of OCB over GCM becomes even more obvious when comparing the throughput/area ratio which is a good indicator for overall efficiency. Figure 14.2a shows the throughput/area ratio for GCM, OCB encryption and OCB-Decryption. GCM encryption and decryption are not listed separately, as they are equal in terms of throughput and area requirements.



(a) Single AE engine.

(b) Overall AE engine.

Figure 14.2: Throughput/Area ratio results for the implemented authenticated encryption engines. Figure 14.2a shows the ratio for single AE architectures and Figure 14.2b displays the results for the AE engines as required by the fast-encryptor system which includes one AE engine for decryption and one for encryption.

Again, it should be noted, that the results between the AES and Serpent architectures cannot be directly compared due to the use of BRAM in AES. It is more meaningful to compare the architectures that are based on the same underlying block cipher. GCM-AES achieves a throughput/area ratio of 4.38 Mbit/ALM while OCB-AES reaches 11.40 Mbit/ALM for encryption and 9.87 Mbit/ALM for decryption. Consequently, OCB, when used with AES is more than twice as efficient as GCM when taking the throughput/area ration as indicator. GCM-Serpent has a ratio of 1.90 Mbit/ALM while OCB-Serpent achieves 4.72 Mbit/ALM for encryption and 4.08 Mbit/ALM for decryption. Again, this is twice as efficient as GCM. Keep in mind, that GCM-AES uses 322 M9K BRAM blocks and that OCB-AES decryption uses 402 M9K BRAM blocks and 2 M144K BRAM blocks while OCB-Serpent decryption requires 3 M144K BRAM blocks. However, the consumption of BRAM blocks is not incorporated in the throughput/area ratio results. Similar results can be observed when looking at the area throughput ratio for the AE engine as required by the fast-encryptor system which includes one AE engine for decryption and one for encryption.

The throughput values given so far are theoretic values that assume, that the pipeline is filled all the time. However, in reality this is not the case. As previously mentioned, key changes as well as *IV* changes can cause pipeline stalls and additional delays. Also, the calculation of the authentication tag is delay in practice. Table 14.4 lists the latency and the different delays that can occur in different authenticated-encryption architectures presented in this work.

Table 14.4: Latency and delays due to key changes and *IV* changes for the different AE architectures.

AE Mode	**Mode**	**Latency** [cycles]	**Delay Key Change** [cycles]	**Delay IV Change** [cycles]	**Delay Tag Calculation** [cycles]
GCM-AES	E/D	4	38	0[a]	4
GCM-Serpent	E/D	4	50	0[a]	4
OCB-AES	E	21	23	0	1
OCB-AES	D	21	23	0	22
OCB-Serpent	E	33	35	0	1
OCB-Serpent	D	33	35	0	33

[a] If an IV of 96 bits is used and the IV is a counter.

For a real life application we assume, that the two communication partners transmit a high amount of data at the highest transmission rate possible. Frames are typically of full size, and key changes are not very frequent. As there are only two partners, a slow key-change rate is realistic. So, when assuming a scenario where a key change happens every 10000$^{\text{th}}$ ethernet frame the negative effect of the delay caused by a key change nearly vanishes. Still, *IV* changes are required for every new frame in GCM and OCB. In GCM, when using a 96-bit *IV* that is a counter, an *IV* change does not cause any additional delay. As our OCB design, as described in Chapter 12 includes measures to effectively reduce the effect of *IV* changes, we were able to reduce the delay caused by IV changes to zero. However, it is necessary that the user applies new *IVs* in advance. In GCM, the authentication tag is ready four cycles after the last ciphertext. This is barely acceptable. While the OCB encryption has only a tag delay of one cycle, OCB decryption has a delay of 22 cycles for AES and 33 cycles for Serpent respectively. As these delays are unacceptably high, the OCB designs feature BRAM blocks that buffer entire ethernet frames until the tag is readily computed. This way, the delayed tag calculation only affects the very first frame being sent after a key change.

14.3 Comparison with Related Work

This section compares the achieved results with related work that also targets high-throughput implementations of block ciphers and authenticated encryption modes. Therefore no publications that focus on low resource consumption are included. In addition we only list publications that provide practical implementations in regard to our target application. This means, that we exclude implementations that rely on precomputed LUTs based on fixed keys and require reprogramming of the FPGA in order to change the secret key. Also note, that we only compare the achieved maximum frequency and throughput.

This has two reasons: first, it is hard to compare the resource consumption on different hardware platforms as the complexity of their CLBs varies and second, this work mainly focused on throughput and not on resource consumption. However, also the achieved throughputs cannot easily be compared due to the different target platforms. The main goal of this section is to give an estimation on how the results achieved in this work perform in comparison with other state-of-the-art implementations. Table 14.5 gives the comparison of encryption-only and authenticated encryption results of the architectures developed in this work with related work.

Table 14.5: Comparison of encryption-only and authenticated encryption results with related work.

Source	**AE Mode**	**Device**	**Cipher Cores**	$\mathbf{f_{max}}$ [MHz]	**Throughput** [Gbit/s]
This work	AES only	Altera EP4S100G5	1	252	32.3
This work	Serpent only	Altera EP4S100G5	1	284	36.3
Sugier [103]	Serpent only	Xilinx XC3S1600E-5	1	137	17.5
Lazaro et al. [64]	Serpent only	Xilinx X2C2000-6	1	333	42.8
This work	GCM-Serpent	Altera EP4S100G5	4	203	104
This work	OCB-AES	Altera EP4S100G5	4	220	112
This work	OCB-Serpent	Altera EP4S100G5	4	267	136
Zhou et al. [113]	GCM-AES	Xilinx V5LX85ff668-2	1	340	40.2
Abdellatif et al. [2]	GCM-AES	Xilinx XC5VLX220	4	200	102.4

Table 14.5 does not contain related work for high-speed AES implementations as the AES cipher was not a main contribution of this work. In 2004 Lázaro et al. [64] presented a Serpent design capable of encrypting 42.8 Gbit/s on an Xilinx Virtex-2 XC2V2000 FPGA. Their architecture is fully unrolled with two pipeline stages for each round of Serpent. In order to increase the throughput the encryption pipeline is clocked at double frequency compared to the key scheduler. While this design uses two pipeline stages per round, and thus is able to outperform the Serpent architecture presented in this work in terms of throughput, this also doubles the latency. Even if this may not be a big disadvantage in general, it is a problem when using the cipher in either GCM or OCB mode, as the delays caused by key changes and IV changes as well as for the tag calculation would double. Sugier [103] describes an implementation of Serpent that uses full outer-loop pipelining and a key scheduler in combinational logic which achieves a throughput of 19.7 Gbit/s targeting a Xilinx Spartan-3E FPGA. A design employing pipelining for the cipher rounds of Serpent as well as for the key scheduler that reaches a throughput of 17.5 Gbit/s is presented in the same paper. Our implementation outperforms this by a factor of two. However, to be fair, the target devices cannot really be compared as the Xilinx Spartan 3 is a low-cost device while the Altera Stratix IV is in the high-end range.

In 2009, Zhou et al. [113] presented a single-core GCM-AES design, which targets a Xilinx Virtex-5 FPGA and achieves a throughput of 41.5 Gbit/s based on the 128-bit version of AES. They use the same pipelined Karatsuba-Ofman multiplier design as applied in this work. In terms of single core performance, they outperform our design when simply dividing the throughput of the parallel GCM design by four. It should however be noted, that the complexity as well as the resource consumption increase when using parallel designs for the encryption and the GHASH. In consequence this also lowers the maximum frequency and the throughput. Abdellatif, Chotin-Avot, and Mehrez [2] describe a GCM-AES design

reaching a throughput of 102.4 Gbit/s on a Virtex-5 FPGA. Their implementation is based four fully unrolled, pipelined AES cores for encryption and four parallel GHASH cores. However, their design uses a fixed secret key which allows them to precalculate the round keys for AES and the H-values for GHASH and to synthesize those into their design. This reduces the complexity for both, the encryption part as well as for the authentication part but requires to reprogram the FPGA in case a key change is needed. Still, our work reaches and a similar throughput for GCM-AES. OCB-Serpent even gives an improvement in throughput of 33 %.

To the best of our knowledge there are no hardware architectures targeting high-throughput authenticated encryption that are based on a block-cipher other than AES. Additionally, no design using AES as a block cipher, that is capable of encrypting at throughputs of over 100 Gbit/s, and that is based on a different mode of operation than GCM has been published so far.

Chapter 15

Conclusion

The contribution of this work is part of the QCrypt project which aims at providing a future-proof secure high-speed communication platform that is based on a quantum key-distribution (QKD) system. The system can be divided into two parts: the quantum key-distribution system and the fast-encryptor system. The fast-encryptor system is based on an Altera Stratix IV EP4S100G5 FPGA and handles the high-speed network communication and generates an authenticated and encrypted data stream using the user-keys provided by the QKD-system. This stream is then transmitted to the communication partner over a public 100 Gbit/s channel. The existing authenticated-encryption engine is based on the Galois Counter Mode of Operation (GCM) with four parallel AES cores as underlying block cipher.

The main goal of this work was to elaborate possible alternatives to the existing authenticated-encryption engine that are capable of encrypting data at rates of over 100 Gbit/s. On the one hand the idea was to be prepared for unexpected security flaws in the existing system by providing alternative subcomponents and on the other hand there was a desire to possibly find a better solution compared to the existing system. In order to do so, we first introduced the theory necessary to understand this thesis and we described the algorithms involved within this thesis. Next, in order to be able to identify promising alternatives we first presented the existing authenticated-encryption engine and analyzed the specifications of the system and requirements the authenticated-encryption engine has to fulfill. Then, we evaluated alternative subcomponents and algorithms in regard to the previously defined requirements. Finally, the most promising candidates were designed, implemented in hardware and tested on the target system. We laid a strong emphasis on ensuring easy integration into the existing system, an efficient use of resources, low latency and minimal delays.

During evaluation of alternatives to the AES block cipher, Serpent emerged as a promising candidate and was consequently implemented in hardware. In order to reach the desired throughput of 100 Gbit/s, 33 pipeline stages had to be introduced and four cipher cores had to be used in parallel. The resulting architecture is able to achieve a throughput of 140 Gbit/s at a maximum frequency of 275 MHz which is an increase of 8.3 % compared to the existing AES architecture. The design requires 33,067 ALMs and does not use any BRAM on the target platform. Based on the multi-core Serpent design, we developed a mode for authenticated encryption that uses GCM with Serpent as the underlying block cipher. This design was able to reach a throughput of 104 Gbit/s at a maximum frequency of

203 MHz while consuming 56,474 ALMs of 212,480 ALMs available on the target platform. In addition the Offset CodeBook (OCB) mode in its third version was chosen as an alternative mode for authenticated encryption. The OCB-encryption architecture is based on four parallel block cipher cores, while the OCB-decryption architecture uses four cipher cores in parallel for decryption and an additional cipher core that is in encryption mode. When using AES as the underlying block cipher OCB achieves a throughput of 112 Gbit/s at a maximum frequency of 220 MHz and only needs 10,060 ALMs for encryption and 11,614 ALMs for decryption. Compared to GCM-AES this is an increase of over 7 % in throughput. The fastest authenticated-encryption design is OCB-Serpent, which reaches a throughput of 136 Gbit/s at 267 MHz while consuming 29,506 ALMs for encryption and 33,891 ALMs for decryption. This is an increase in throughput of 27 % compared to GCM-AES. When looking at the throughput/area ratio OCB is more than twice as efficient as GCM.

To summarize, this work produced a multi-core Serpent architecture based on four parallel cipher cores capable of encrypting at a throughput of 140 Gbit/s. In addition, we implemented three authenticated-encryption engine variants that are all capable of providing authenticated encryption at a rate of over 100 Gbit/s: *GCM-Serpent*, *OCB-AES* and *OCB-Serpent*. To the best of our knowledge, GCM-Serpent and OCB-Serpent are the first hardware architectures targeting high-throughput authenticated encryption that are based on a block-cipher other than AES. Additionally, no design using AES as a block cipher, that is capable of reaching throughputs of over 100 Gbit/s, and that is based on a different mode of operation than GCM has been published so far. Our fastest design is based on OCB-Serpent and reaches a throughput of 136 Gbit/s which outperforms all GCM-AES implementations available on FPGAs to date. Furthermore, we could show, that OCB is twice as efficient compared to GCM when taking the throughput/area ratio as an indicator.

15.1 Future Work

This section gives some possible tasks that could improve some aspects of the existing architectures.

Implement AES S-Boxes in LUTs It could be beneficial to implement the AES S-Boxes using a LUT-based approach. Modern high-end FPGAs like the Altera Stratix IV EP4S100G5 allow to implement the 8-bit S-Boxes of AES relatively efficiently. Results in [65, 87] suggest, that this approach could result in an increased throughput for AES and consequently also an increased throughput for OCB-AES.

AES with Shared Encryption and Decryption Data Path An AES with shared data path for encryption and decryption would be beneficial for OCB. Using four cipher cores that are capable of encrypting and decrypting would improve the flexibility of the system. It would allow to authenticate data at the decryption side at the same speed as on the encryption side. In addition, this would allow the network interface to apply messages to the authenticated-encryption engine exactly in the order it arrive as authenticated only data can be processed as fast as authenticated and encrypted data. As AES can efficiently share resources between encryption and decryption and the additional encryption core

included in the current architecture could be omitted, the resource consumption should not increase drastically.

Evaluate CEASAR Challenge Submissions The CAESAR challenge [27] generated a variety of new authenticated-encryption algorithms. These submissions should be evaluated towards their suitability for high-throughput implementations. Still, as the challenge is in an early stage, the security of some schemes is still unclear. Thus, the employment of the novel proposals in commercial products such as the QCrypt should be chosen with great care.

15.2 Outlook

In 2013 the CAESAR challenge [27] was launched, with the goal to identify authenticated ciphers that offer advantages over GCM-AES and that are suitable for widespread adoption. The whole competition is based on a public evaluation and discussion. For the first round of the competition, launched in March 2014, 57 authenticated cipher candidates were submitted. The candidates are now under public review and their security is deeply analyzed. At the time of writing this thesis eight submissions have been withdrawn because of security flaws that could be identified. In addition to the security research, the suitability of the algorithms implementations on different platforms is evaluated. Software implementations of the AE algorithms are compared on different target processors and evaluated regarding throughput, energy efficiency, and memory requirements. As requirements vary for different target applications, the evaluations are performed for a wide range of platforms. Some, that target embedded systems or low-energy environments, focus on the resource requirements and the energy efficiency, while others that target more capable processors mainly opt for high throughput. The very same is true for hardware implementations targeting FPGAs or ASICs. One the one hand the submissions are implemented and evaluated with low resource consumption and high energy efficiency in mind, and on the other hand high throughput is the main objective. Typically, most evaluations compare the throughput/area ratio of the implemented candidates as it is a good tool to measure the overall efficiency of an algorithm. Based on the observation and findings made in the first round, in January 2015, submission will be announced that make it through to the second round of the CAESAR challenge. These submissions will again be analyzed and evaluated in order to select candidates that make it to the third round, which starts in December 2015. The process of evaluation is then repeated once more until in December 2016 the submissions that make it to the final round of the CAESAR challenge will be announced. One, or multiple of these final candidates will then, in December 2017, be announced as winners of the competition.

Still, at the time of writing this work, the CAESAR challenge is in an very early stage. It can not yet be estimated which candidates will make the run. It will be interesting to see which algorithms prove to be highly secure and particularly well suited for hardware implementations, especially targeting high throughput.

Appendix A

Definitions

A.1 Abbreviations

AE	Authenticated Encryption
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
ALM	Arithmetic Logic Module
ALUT	Adaptive Look-Up Table
ASIC	Application-Specific Integrated Circuit
BRAM	Block Random Accessible Memory
CPLD	Complex Programmable Logic Device
CRC	Cyclic Redundancy Checksum
DES	Data Encryption Standard
DUT	Device Under Test
ECB	Electronic Codebook Mode
EEPROM	Electrically Erasable Programmable ROM
EOF	End Of Frame
FIPS	Federal Information Processing Standard
FPGA	Field Programmable Array
GCM	Galois Counter Mode of Operation
HDL	Hardware Description Language
I/O	Input/Output
IV	Initial Vector
KOA	Karatsuba-Ofman Algorithm
LE	Logic Element
LUT	Look-Up Table
MAC	Message Authentication Code
MDC	Modification Detection Code
MoO	Mode of Operation
MSB	Most Significant Bit
MUX	Multiplexer
NIST	National Institute of Standardization and Technology
ntz	Number of Trailing Zeros
OCB	Offset Codebook Mode of Operation

OTP	One-Time Pad
PCB	Printed Circuit Board
PKI	Public-Key Infrastructure
PLL	Phase-Locked Loop
PLD	Programmable Logic Device
PROM	Programmable Read-Only Memory
QC	Quantum Cryptography
QKD	Quantum Key Distribution
ROM	Read-Only Memory
SRAM	Static Random-Access Memory
SOF	Start Of Frame

A.2 Used Symbols

$\|$	Concatenation
$\oplus$	Bitwise-exclusive-or
$\wedge$	Bitwise-and-operation
$\ll$	Left shift
$\lll$	Circular left shift
$\ggg$	Circular right shift
$\varnothing$	Empty set or empty string
$\lvert x \rvert$	Length of string x in bits
$msb(x)$	Returns the most significant bit of x using binary representation

Appendix B

Supplementary Material

B.1 Serpent

S-Boxes and Inverse S-Boxes

The S-Boxes S_0 through S_7 used for encryption in Serpent are defined as follows:

S_0 :	3	8	15	1	10	6	5	11	14	13	4	2	7	0	9	12
S_1 :	15	12	2	7	9	0	5	10	1	11	14	8	6	13	3	4
S_2 :	8	6	7	9	3	12	10	15	13	1	14	4	0	11	5	2
S_3 :	0	15	11	8	12	9	6	3	13	1	2	4	10	7	5	14
S_4 :	1	15	8	3	12	0	11	6	2	5	4	10	9	14	7	13
S_5 :	15	5	2	11	4	10	9	12	0	3	14	8	13	6	7	1
S_6 :	7	2	12	5	8	4	6	11	14	9	1	15	13	3	10	0
S_7 :	1	13	15	0	14	8	2	11	7	4	12	10	9	3	5	6

The inverse S-Boxes InvS_0 through InvS_7 used for decryption in Serpent are defined as follows:

InvS_0 :	13	3	11	0	10	6	5	12	1	14	4	7	15	9	8	2
InvS_1 :	5	8	2	14	15	6	12	3	11	4	7	9	1	13	10	0
InvS_2 :	12	9	15	4	11	14	1	2	0	3	6	13	5	8	10	7
InvS_3 :	0	9	10	7	11	14	6	13	3	5	12	2	4	8	15	1
InvS_4 :	5	0	8	3	10	9	7	14	2	12	11	6	4	15	13	1
InvS_5 :	8	15	2	9	4	1	13	14	11	6	5	3	7	12	10	0
InvS_6 :	15	10	1	13	5	3	6	0	4	9	14	7	2	12	8	11
InvS_7 :	3	0	6	13	9	14	15	8	5	12	11	7	10	1	4	2

Initial Permutation and Final Permutation

The Initial and Final permutations are each represented by an array containing the integers in $0 \ldots 127$ without repetitions. Having a value v (say, 32) at position p (say, 1) means that the output bit at position $p(1)$ comes from the input bit at position $v(32)$.

Initial Permutation

0	32	64	96	1	33	65	97	2	34	66	98	3	35	67	99
4	36	68	100	5	37	69	101	6	38	70	102	7	39	71	103
8	40	72	104	9	41	73	105	10	42	74	106	11	43	75	107
12	44	76	108	13	45	77	109	14	46	78	110	15	47	79	111
16	48	80	112	17	49	81	113	18	50	82	114	19	51	83	115
20	52	84	116	21	53	85	117	22	54	86	118	23	55	87	119
24	56	88	120	25	57	89	121	26	58	90	122	27	59	91	123
28	60	92	124	29	61	93	125	30	62	94	126	31	63	95	127

Final Permutation

0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
64	68	72	76	80	84	88	92	96	100	104	108	112	116	120	124
1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61
65	69	73	77	81	85	89	93	97	101	105	109	113	117	121	125
2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62
66	70	74	78	82	86	90	94	98	102	106	110	114	118	122	126
3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63
67	71	75	79	83	87	91	95	99	103	107	111	115	119	123	127

Serpent Key Scheduler Architecture

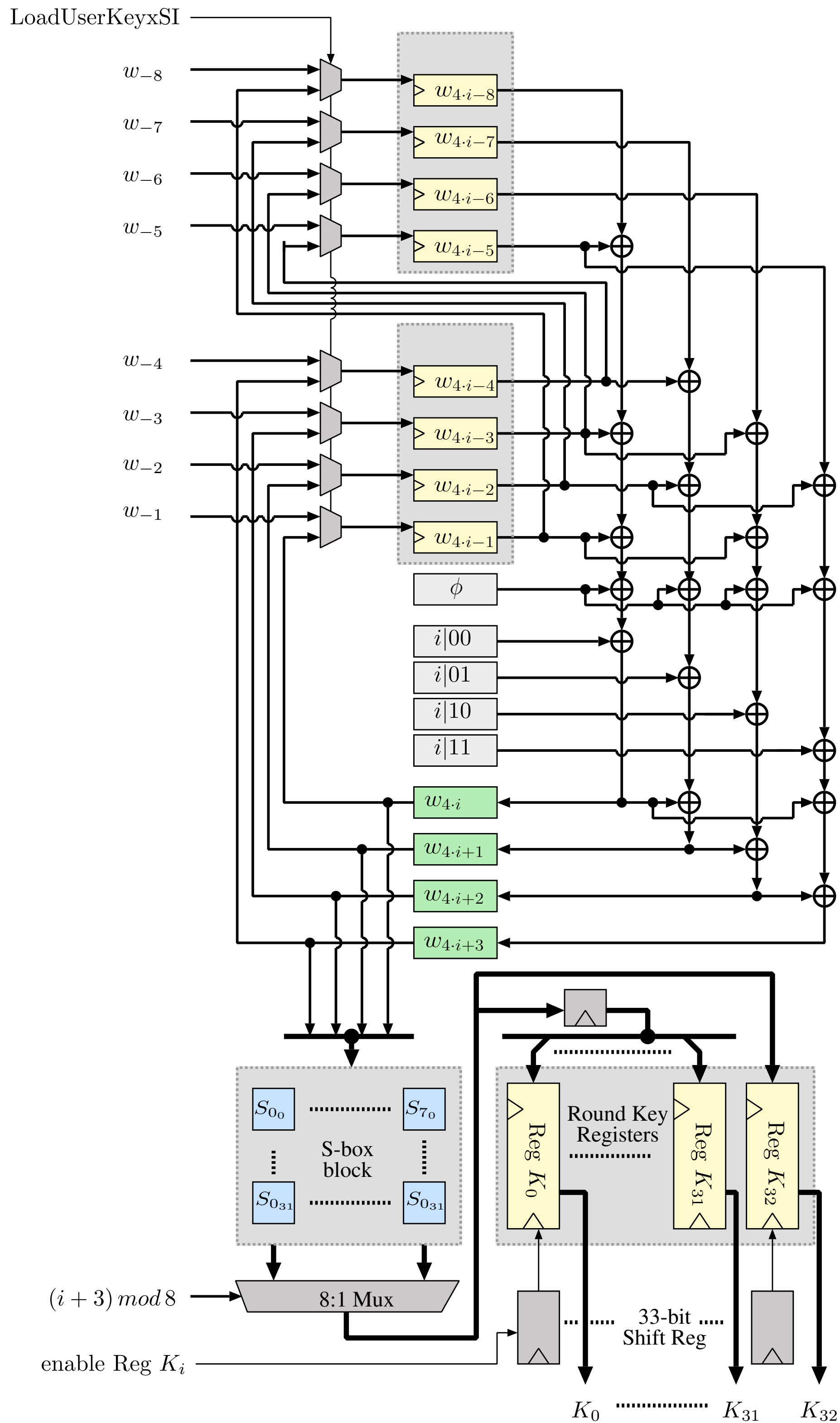


Figure B.1: Overview of the key-scheduler design of Serpent.

B.2 OCB

Algorithm B.2.1 gives an exact description of the authenticated decryption according to OCB. The characters $\|$, $\oplus$, and $\wedge$ denote the concatenation-, bitwise-exclusive-or-, and bitwise-and-operation, respectively. Furthermore, *ntz(i)* describes the number of trailing zeroes of i in binary representation. We use $\varnothing$ to represent both an empty binary string of length n (cf. line 3), and an *empty set* as within line 11. The listings for the procedures *Setup*, *Init*, and $Hash_K$ used throughout Algorithm B.2.1 are equal to the OCB authenticated encryption and are given in Algorithm 6.6.2, Algorithm 6.6.3 and in Algorithm 6.6.4.

Algorithm B.2.1 OCB authenticated decryption.

Input: Ciphertext C of m blocks length, Message block length n, Cipher key K, Nonce N, Associated data A of p blocks length, Tag length τ, Authentication tag T

Output: Plaintext M, Authentication tag valid $\in \{VALID, INVALID\}$

1: **if** $|N| \geq n$ **then return** INVALID
2: $\{C_1, \ldots, C_m, C_*\} \leftarrow C$, with $|C_i| = n$ and $|C_*| < n$
3: $Checksum \leftarrow \varnothing; M \leftarrow \varnothing$
4: $L_*, L_\$, L[0] \ldots L[\lfloor log_2(m) \rfloor] \leftarrow Setup(K, m)$
5: $\Delta \leftarrow Init(N, n, K)$
6: **for** $i = 1$ to m **do**
7: $\quad \Delta \leftarrow \Delta \oplus L[ntz(i)]$
8: $\quad M \leftarrow M \,\|\, D_K(C_i \oplus \Delta) \oplus \Delta$
9: $\quad Checksum \leftarrow Checksum \oplus M_i$
10: **end for**
11: **if** $M_* \neq \varnothing$ **then**
12: $\quad \Delta \leftarrow \Delta \oplus L_*$
13: $\quad Pad \leftarrow E_K(\Delta)$
14: $\quad M \leftarrow M \,\|\, C_* \oplus (Pad \wedge (2^{|C_*|} - 1))$
15: $\quad Checksum \leftarrow Checksum \oplus M_* 10^*$, with
$\qquad M_* 10^* = M_* \| 1 \| 0 \ldots 0$, such that $|M_* 10^*| = n$
16: **end if**
17: $\Delta \leftarrow \Delta \oplus L_\$$
18: $Final \leftarrow E_K(Checksum \oplus \Delta)$
19: $Auth \leftarrow Hash_K(A)$
20: $Tag \leftarrow Final \oplus Auth$
21: $T' \leftarrow Tag \wedge (2^\tau - 1)$
22: **if** $T == T'$ **then**
23: $\quad$ **return** $VALID$
24: $\quad$ **return** M
25: **else**
26: $\quad$ **return** $INVALID$
27: **end if**

Appendix C

Original Assignment

The original assignment of this thesis is appended in the next five pages. It comprises an introduction to the QCrypt project and its goals and points out the need for a 100 Gbit/s authenticated-encryption engine. Next follows a general description of the project which involves two main tasks. First, possible alternatives to the existing AE engine have to be evaluated and second, one the hardware design of one promising alternative AE engine has to be implemented on an Stratix IV FPGA and an evaluation of its performance has to be carried out. The implementation has to be compatible to the existing system. Next follows a list of goals and milestones, to reach these goals are set that have to be fulfilled in sequential order. First, algorithms suited for the requirements of the system have to be found and evaluated. Second, the chosen algorithms have to be implemented and finally the have to be evaluated. At the end, a set of requirements to realize the project is given. This set includes the preparation of a project plan, weekly meetings, a report and a presentation which both present the results of the work and a listing of deliverables required to successfully finish the work.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Master Thesis at the Departement of
Information Technology and Electrical Engineering

Winter Term 2011

Christian Pendl

Block-cipher- and authentification-algorithms for very high speed communication system

September 16, 2011

Advisors: Christoph Keller (IIS), ETZ J69.2, Tel. 632 54 33, chrikell@iis.ee.ethz.ch
Michael Mühlberghuber, ETZ J69.2, Tel. ??????, ??@iis.ee.ethz.ch
Handout: September 19, 2011
Due: March 19 , 2012

The final report will be turned in electronic format. All copies remain property of the Integrated Systems Laboratory.

1 Introduction

The QCrypt Project aims to considerably improve cryptography on both the key distribution level and the encryption level. Quantum Key Distribution is a secure way to generate keys, which is based on the fundamental laws of quantum mechanics. However, existing systems are too slow. The new QKD system will be capable of producing keys at 1Mb/s rate, which means it will allow 1MHz OTP encryption for high-level applications. In standard applications the data exchange rates continue to increase.

Today's commercial encryptors are already approaching 10Gb/s. Consequently a future proof encryption engine for up to 100 Gb/s will be developed and this high-speed encryption will be combined with high rate QKD, to allow for changing the keys rapidly, thus considerably improving the security and simplifying the key management. [2]

The encryption engine will be capable of encrypting at 100 Gb/s as well as authenticating the message at said speed. Different modes of encryption and authentication are supported. The whole Ethernet packet can be encrypted and encapsulated within another Ethernet packet, only payload can be encrypted, or the whole frame can be transferred in plain text, and all modes can either be authenticated or not.

2 Project Description

At the moment, the encryption is done by a 4 parallel AES cores in counter mode. The authentication is realized through a GHASH core. The goal of this project is the development of a high-speed cryptographic encryption module, compatible with the current QCrypt-environment. This includes

- Evaluation of different authenticated encryption modes of operation
  - GCM
  - OCB
  - CWC (Carter Wegman with Counter)
  - EAX
  - CBC-MAC (CCM)
  - ...
- Evaluation and comparison of different block-cipher candidates
  - Serpent
  - Twofish
  - AES
  - ...
- Evaluation of different (universal) hash functions (and SHA-3 candidates)
  - GHASH
  - Gröstl

    - Keccak
    - ...

- Hardware Design and evaluation of performance
    - Encryption/decryption operation
        * Pipelined data path
        * Parallel GF(2^m) binary multiplier for GHASH
    - Message authentication

2.1 Interface

The data interface of the encryption / authentication module should be compatible with the currently used interfaces. Currently, a 2 x 512 bit wide interface is used at either end, 512 bit for the plain data, 512 bit for the cipher text.

The Key and some settings are stored in registers which are accessed through a USB connection. Further control signals can be used if needed.

2.2 Hardware Platform

The hardware platform is based on a newly developed PCB, currently under testing, hosting a Stratix IV GT (EP4S100G5) [3] FPGA with 32 10 Gb/s serial transceiver, 531'200 equivalent Logic Elements (LE) and about 18'000 Kibits of embedded memory. Further a Max II CPLD (EPM2210) is used for FPGA configuration and hardware monitoring.

The communication can be realized by up to 8 SFP+ and 2 XFP modules for the plain text side and either a CXP or a CFP module at the cipher text side. CXP is a cheap variant for 100 Gb/s communication realized in so called active cables with 20 parallel fibers and the optical modules located in the plug whereas CFP is a communication of 100 Gb/s over one single fiber.

3 Goals

This project will not be considered a success unless a working demonstration can be presented at the end of the project. This system should:

- be capable of encrypting and authenticating data streams at a minimal rate of 100 Gb/s
- be optimized for low area consumption
- have a validated data path.

4 Milestones

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

- **System specification**
  The first task is to determine the specifications of the system. This includes usable logic, throughput, latency and controlling.

- **Algorithm evaluation**
  For the system, suited algorithms have to be found. These algorithms need to be analyzed, simulated, and compared. Especially the throughput and complexity need to be compared thoroughly.

- **Implementation**
  The chosen algorithms will then be implemented and optimized for the target FPGA. Pipelining and Retiming will be needed to reach the goals of throughput.

- **Functional test**
  Two different functional test should be done. One is a standalone test of the crypto system. Here, no interaction with the networking components will be necessary. The results of the operation need to be validated by a golden model. The second test consists of a run time test with actual interaction with the Ethernet circuitry.

5 Project Realization

5.1 Project Plan

Within the first month of the project you will be asked to prepare a project plan. This plan should identify the tasks to be performed during the project and set deadlines for those tasks. The prepared plan will be a topic of discussion of the first week's meeting between the students and the advisors. Note that the project plan should be updated constantly depending on the project's status.

5.2 Meetings

Weekly meetings will be held between the student and the assistants at a time to be determined. These meetings will be used to evaluate the status and progress of the project. In addition to these regular meetings, additional meetings can be organized to address urgent issues as well.

5.3 Reports

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

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

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

Final Report The final report has to be presented at the end of the Master Thesis and a digital copy needs to be handed out and remains property of the IIS. This report is only accepted when the keys for the ETZ building have been properly returned. Note that this task description is part of your thesis and has to be attached to your final report.

5.4 Presentation

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

References

[1] R Anderson, E. Biham, L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard", https://www.cl.cam.ac.uk/ rja14/serpent.html, 1998

[2] QCRYPT : "Secure High-Speed Communication based on Quantum Key Distribution" , http://www.nano-tera.ch/nanoterawiki/QCRYPT

[3] Altera, "Stratix IV Handbook", http://www.altera.com/literature/hb/stratix-iv/stratix4_handbook.pdf

[4] H. Kaeslin, "Digital Integrated Circuit Design", Cambridge University Press, 2008

[5] Design Zentrum website: `http://www.dz.ee.ethz.ch` and VHDL naming conventions: `http://www.dz.ee.ethz.ch/en/information/hdl-help/vhdl-naming-conventions.html`

Zurich, September 16, 2011 Prof. Dr. Hubert Kaeslin

The thesis will not be accepted without returning the keys!

Bibliography

[1] K.M. Abdellatif, R. Chotin-Avot, and H. Mehrez. “High Speed Authenticated Encryption for Slow Changing Key Applications Using Reconfigurable Devices”. In: *Wireless Days (WD), 2013 IFIP*. Nov. 2013, pp. 1–6. DOI: `10.1109/WD.2013.6686460` (cit. on p. 3).

[2] K.M. Abdellatif, R. Chotin-Avot, and H. Mehrez. “Improved Method for Parallel AES-GCM Cores Using FPGAs”. In: *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*. Dec. 2013, pp. 1–4. DOI: `10.1109/ReConFig.2013.6732299` (cit. on pp. 3, 94).

[3] *AES Competition*. 1997. URL: `http://competitions.cr.yp.to/aes.html` (visited on 10/29/2014) (cit. on p. 3).

[4] Martin Ågren et al. “Grain-128a: a New Version of Grain-128 with Optional Authentication”. eng. In: International Journal of Wireless and Mobile Computing 5.1 (2011), pp. 48–59. ISSN: 1741-1084 (cit. on p. 33).

[5] Abdulkadir Akin et al. “Efficient Hardware Implementations of High Throughput SHA-3 Candidates Keccak, Luffa and Blue Midnight Wish for Single- and Multi-Message Hashing”. In: *Proceedings of the 3rd international conference on Security of information and networks*. SIN ’10. Taganrog, Rostov-on-Don, Russian Federation: ACM, 2010, pp. 168–177. ISBN: 978-1-4503-0234-0 (cit. on p. 65).

[6] Liakot Ali et al. “Design of an Ultra High Speed AES Processor for Next Generation IT Security”. In: *Computers & Electrical Engineering* 37.6 (2011), pp. 1160–1170. ISSN: 0045-7906. DOI: `http://dx.doi.org/10.1016/j.compeleceng.2011.06.003` (cit. on pp. 2, 62).

[7] R. Alleaume et al. “SECOQC White Paper on Quantum Key Distribution and Cryptography”. In: *eprint arXiv:quant-ph/0701168* (Jan. 2007). eprint: `arXiv:quant-ph/0701168` (cit. on pp. 19, 20).

[8] Basel Alomair, Andrew Clark, and Radha Poovendran. “The Power of Primes: Security of Authentication Based on a Universal Hash-Function Family”. English. In: *Journal of Mathematical Cryptology* 4.2 (2010), pp. 121–148 (cit. on p. 65).

[9] Altera Corp. *Stratix IV Device Handbook Volume 1*. Sept. 2012 (cit. on pp. 50, 51, 80).

[10] R. Anderson, E. Biham, and L. Knudsen. "Serpent: A Proposal for the Advanced Encryption Standard". In: *Proceedings of the First AES Candidate Conference*. National Institute of Standard and Technology. Ventura, CA, USA, June 1998 (cit. on pp. 2, 23, 27, 28).

[11] H. Bechmann-Pasquinucci and N. Gisin. "Incoherent and Coherent Eavesdropping in the Six-State Protocol of Quantum Cryptography". In: *Phys. Rev. A* 59 (6 June 1999), pp. 4238–4248. DOI: 10.1103/PhysRevA.59.4238 (cit. on p. 18).

[12] M. Bellare, P. Rogaway, and D. Wagner. *EAX: A Conventional Authenticated-Encryption Mode*. Cryptology ePrint Archive, Report 2003/069. 2003 (cit. on p. 65).

[13] Mihir Bellare and Chanathip Namprempre. "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm". In: *Advances in Cryptology — ASIACRYPT 2000*. Ed. by Tatsuaki Okamoto. Vol. 1976. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, Oct. 2000. Chap. 41, pp. 531–545. ISBN: 978-3-540-41404-9 (cit. on pp. 33, 65).

[14] Mihir Bellare and Phillip Rogaway. "Introduction to Modern Cryptography". In: *UCSD CSE 207 Course Notes*. 2005, p. 207 (cit. on p. 10).

[15] Mihir Bellare, Phillip Rogaway, and David Wagner. "The EAX Mode of Operation". In: *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*. Ed. by Bimal K. Roy and Willi Meier. Vol. 3017. Lecture Notes in Computer Science. Springer, Feb. 2004, pp. 389–407. DOI: 10.1007/b98177 (cit. on p. 65).

[16] Charles H. Bennett. "Quantum Cryptography Using any Two Nonorthogonal States". In: *Phys. Rev. Lett.* 68 (21 May 1992), pp. 3121–3124. DOI: 10.1103/PhysRevLett.68.3121 (cit. on p. 17).

[17] Charles H. Bennett, Gilles Brassard, and N. David Mermin. "Quantum Cryptography Without Bell's Theorem". In: *Phys. Rev. Lett.* 68 (5 Feb. 1992), pp. 557–559. DOI: 10.1103/PhysRevLett.68.557 (cit. on p. 18).

[18] Charles H Bennett, Gilles Brassard, et al. "Quantum Cryptography: Public Key Distribution and Coin Tossing". In: *Proceedings of IEEE International Conference on Computers, Systems and Signal Processing*. Vol. 175. Bangalore, India. 1984 (cit. on pp. 15, 16).

[19] C.H. Bennett et al. "Generalized Privacy Amplification". In: *Information Theory, IEEE Transactions on* 41.6 (Nov. 1995), pp. 1915–1923. ISSN: 0018-9448. DOI: 10.1109/18.476316 (cit. on p. 21).

[20] G. Bertoni et al. *The Keccak SHA-3 Submission*. Submission to NIST (Round 3). 2011 (cit. on p. 65).

[21] E. Biham. "A Fast New DES Implementation in Software". In: *Fast Software Encryption*. Springer. 1997, pp. 260–272 (cit. on p. 27).

[22] Nikita Borisov, Ian Goldberg, and David Wagner. "Intercepting Mobile Communications: The Insecurity of 802.11". In: *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*. MobiCom '01. Rome, Italy: ACM, 2001, pp. 180–189. ISBN: 1-58113-422-3. DOI: 10.1145/381677.381695 (cit. on p. 34).

[23] Gilles Brassard and Louis Salvail. "Secret-Key Reconciliation by Public Discussion". English. In: *Advances in Cryptology — EUROCRYPT '93*. Ed. by Tor Helleseth. Vol. 765. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1994, pp. 410–423. ISBN: 978-3-540-57600-6. DOI: `10.1007/3-540-48285-7_35` (cit. on p. 17).

[24] Dagmar Bruss. "Optimal Eavesdropping in Quantum Cryptography with Six States". In: *Phys. Rev. Lett.* 81 (14 Oct. 1998), pp. 3018–3021. DOI: `10.1103/PhysRevLett.81.3018` (cit. on p. 18).

[25] K. Burgin and M. Peck. *Suite B Profile for Internet Protocol Security (IPsec)*. RFC 6380 (Informational). Internet Engineering Task Force, Oct. 2011 (cit. on p. 3).

[26] Carolynn Burwick et al. "MARS - a Candidate Cipher for AES". In: *NIST AES Proposal* (1999) (cit. on pp. 2, 23).

[27] *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness.* Mar. 2013. URL: `http://competitions.cr.yp.to/caesar.html` (visited on 10/29/2014) (cit. on pp. 3, 32, 98).

[28] Xin Cai, Rong Sun, and Jingwei Liu. "An Ultrahigh Speed AES Processor Method Based on FPGA". In: *Intelligent Networking and Collaborative Systems (INCoS), 2013 5th International Conference on.* Sept. 2013, pp. 633–636. DOI: `10.1109/INCoS.2013.123` (cit. on pp. 2, 62).

[29] J. Crenne et al. "Efficient Key-Dependent Message Authentication in Reconfigurable Hardware". In: *Field-Programmable Technology (FPT), 2011 International Conference on.* Dec. 2011, pp. 1–6. DOI: `10.1109/FPT.2011.6132722` (cit. on p. 4).

[30] *Crypto++® Library 5.6.2.* Available online: `http://www.cryptopp.com/`. 2010 (cit. on p. 83).

[31] Joan Daemen and Vincent Rijmen. *AES Proposal: Rijndael, AES algorithm submission, September 3, 1999.* 1999. URL: `http://www.nist.gov/CryptoToolKit` (cit. on pp. 2, 23).

[32] Joan Daemen and Vincent Rijmen. *The Design of Rijndael.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2002. ISBN: 3540425802 (cit. on pp. 11, 24, 26).

[33] Andreas Dandalis, Viktor K. Prasanna, and Jose D. P. Rolim. *A Comparative Study of Performance of AES Final Candidates Using FPGAs.* 2000 (cit. on pp. 63, 64).

[34] W. Diffie and M. Hellman. "New Directions in Cryptography". In: *Information Theory, IEEE Transactions on* 22.6 (1976), pp. 644–654 (cit. on pp. 20, 21).

[35] Christoph Dobraunig et al. *ASCON v1: Submission to the CAESAR Competition.* Available online: `http://competitions.cr.yp.to/round1/asconv1.pdf`. Mar. 2014 (cit. on pp. 3, 65).

[36] M. Dušek, N. Lütkenhaus, and M. Hendrych. "Quantum cryptography". In: *Progress in Optics* 49 (2006), pp. 381–454 (cit. on pp. 17, 19).

[37] Artur K. Ekert. "Quantum Cryptography Based on Bell's Theorem". In: *Physical Review Letters* 67.6 (Aug. 5, 1991), pp. 661–663. DOI: `10.1103/physrevlett.67.661` (cit. on p. 18).

[38] AJ Elbirt and C Paar. "An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher". In: *EIGHTH ACM INTERNATIONAL SYMPOSIUM ON FIELD-PROGRAMMABLE GATE ARRAYS*. 2000, pp. 33–40 (cit. on pp. 63, 64).

[39] AJ Elbirt et al. "An FPGA-Based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists". In: *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS* 9 (2001), pp. 545–557 (cit. on pp. 63, 64).

[40] Daniel Engels et al. "The Hummingbird-2 Lightweight Authenticated Encryption Algorithm". English. In: *RFID. Security and Privacy*. Ed. by Ari Juels and Christof Paar. Vol. 7055. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 19–31. ISBN: 978-3-642-25285-3. DOI: `10.1007/978-3-642-25286-0_2` (cit. on p. 33).

[41] Reza Rezaeian Farashahi, Bahram Rashidi, and Sayed Masoud Sayedi. "FPGA Based Fast and High-Throughput 2-Slow Retiming 128-bit AES Encryption Algorithm". In: *Microelectronics Journal* 45.8 (2014), pp. 1014–1025. ISSN: 0026-2692. DOI: `http://dx.doi.org/10.1016/j.mejo.2014.05.004` (cit. on p. 3).

[42] Niels Ferguson et al. "Helix: Fast Encryption and Authentication in a Single Cryptographic Primitive". English. In: *Fast Software Encryption*. Ed. by Thomas Johansson. Vol. 2887. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 330–346. ISBN: 978-3-540-20449-7. DOI: `10.1007/978-3-540-39887-5_24` (cit. on p. 33).

[43] PUB FIPS. "46-3: Data Encryption Standard (DES)". In: *National Institute of Standards and Technology* 25.10 (1999) (cit. on pp. 2, 23).

[44] Ewan Fleischmann, Christian Forler, and Stefan Lucks. "McOE: A Family of Almost Foolproof On-Line Authenticated Encryption Schemes". English. In: *Fast Software Encryption*. Ed. by Anne Canteaut. Vol. 7549. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 196–215. ISBN: 978-3-642-34046-8. DOI: `10.1007/978-3-642-34047-5_12` (cit. on p. 34).

[45] Kris Gaj and Pawel Chodowiec. "Fast Implementation and Fair Comparison of the Final Candidates for Advanced Encryption Standard Using Field Programmable Gate Arrays". English. In: *Topics in Cryptology — CT-RSA 2001*. Ed. by David Naccache. Vol. 2020. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 84–99. ISBN: 978-3-540-41898-6. DOI: `10.1007/3-540-45353-9_8` (cit. on pp. 63, 64).

[46] Kris Gaj et al. "Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs." In: *IACR Cryptology ePrint Archive* 2012 (2012). informal publication, p. 368 (cit. on p. 65).

[47] Nicolas Gisin et al. "Quantum Cryptography". In: *Rev. Mod. Phys* 74 (2002), pp. 145–195 (cit. on pp. 15, 17, 19).

[48] Daniel Gottesman et al. "Security of Quantum Key Distribution with Imperfect Devices". In: *Quantum Info. Comput.* 4.5 (Sept. 2004), pp. 325–360. ISSN: 1533-7146 (cit. on p. 18).

[49] L. Henzen and W. Fichtner. "FPGA Parallel-Pipelined AES-GCM Core for 100G Ethernet Applications". In: *ESSCIRC, 2010 Proceedings of the*. IEEE, Sept. 2010, pp. 202–205. DOI: `10.1109/ESSCIRC.2010.5619894` (cit. on pp. 3, 52, 62).

[50] Ekawat Homsirikamol, Marcin Rogawski, and Kris Gaj. *Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs*. Cryptology ePrint Archive, Report 2010/445. 2010 (cit. on p. 65).

[51] Ekawat Homsirikamol, Marcin Rogawski, and Kris Gaj. "Throughput vs. Area Trade-offs in High-Speed Architectures of Five Round 3 SHA-3 Candidates Implemented Using Xilinx and Altera FPGAs". In: *Proceedings of the 13th international conference on Cryptographic hardware and embedded systems*. CHES'11. Nara, Japan: Springer-Verlag, 2011, pp. 491–506. ISBN: 978-3-642-23950-2 (cit. on p. 65).

[52] IEEE. *IEEE Standard for Information Technology — Telecommunications and Information Exchange Between Systems — Local and Metropolitan Area Networks-Specific Requirements Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications Amendment 4: Media Access Control Parameters, Physical Layers and Management Parameters for 40 Gb/s and 100 Gb/s Operation*. Ed. by Ilango S. Ganga et al. New York, NY, USA: LAN/MAN Standards Committee, June 22, 2010. DOI: `10.1109/ieeestd.2010.5501740` (cit. on p. 59).

[53] IETF. *Counter with CBC-MAC (CCM)*. RFC 3610 (Informational). Internet Engineering Task Force, Sept. 2003 (cit. on pp. 35, 65).

[54] K. Igoe. *Suite B Cryptographic Suites for Secure Shell (SSH)*. RFC 6239 (Informational). Internet Engineering Task Force, May 2011 (cit. on p. 3).

[55] Charanjit Jutla. "Encryption Modes with Almost Free Message Integrity". In: *Advances in Cryptology — EUROCRYPT 2001*. Ed. by Birgit Pfitzmann. Vol. 2045. Lecture Notes in Computer Science. 10.1007/3-540-44987-6_32. Springer Berlin / Heidelberg, 2001, pp. 529–544. ISBN: 978-3-540-42070-5 (cit. on p. 39).

[56] Hubert Kaeslin. *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication*. 1st. New York, NY, USA: Cambridge University Press, 2008. ISBN: 0521882672, 9780521882675 (cit. on pp. 44, 59).

[57] A. Kerckhoffs. "La Cryptographie Militaire". In: *Journal des Sciences Militaires* (1883), pp. 161–191 (cit. on p. 9).

[58] Tadayoshi Kohno. "Attacking and Repairing the winZip Encryption Scheme". In: *Proceedings of the 11th ACM Conference on Computer and Communications Security*. CCS '04. Washington DC, USA: ACM, 2004, pp. 72–81. ISBN: 1-58113-961-6. DOI: `10.1145/1030083.1030095` (cit. on p. 34).

[59] Tadayoshi Kohno, John Viega, and Doug Whiting. *CWC: A High-Performance Conventional Authenticated Encryption Mode*. Cryptology ePrint Archive, Report 2003/106. 2003 (cit. on pp. 32, 36, 65).

[60] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-Hashing for Message Authentication*. United States, 1997 (cit. on p. 14).

[61] T. Krovetz and P. Rogaway. *The OCB Authenticated-Encryption Algorithm*. RFC 7253 (Informational). Internet Engineering Task Force, May 2014 (cit. on pp. 39, 65).

[62] Ted Krovetz and Phillip Rogaway. *OCB (v1). CAESAR First Round Submission.* Online: http://competitions.cr.yp.to/round1/ocbv1.pdf. 2014 (cit. on pp. 39, 65, 67).

[63] Ted Krovetz and Phillip Rogaway. "The Software Performance of Authenticated-Encryption Modes". In: *Proceedings of the 18th International Conference on Fast Software Encryption.* FSE'11. Lyngby, Denmark: Springer-Verlag, 2011, pp. 306–327. ISBN: 978-3-642-21701-2 (cit. on pp. 33, 39, 65).

[64] Jesús Lázaro et al. "High Throughput Serpent Encryption implementation". In: *Field Programmable Logic and Application.* Springer, 2004, pp. 996–1000 (cit. on pp. 3, 63, 64, 70, 94).

[65] Qiang Liu, Zhenyu Xu, and Ye Yuan. "A 66.1 Gbps Single-pipeline AES on FPGA". In: *Field-Programmable Technology (FPT), 2013 International Conference on.* Dec. 2013, pp. 378–381. DOI: 10.1109/FPT.2013.6718392 (cit. on pp. 2, 62, 97).

[66] U.M. Maurer. "Secret Key Agreement by Public Discussion from Common Information". In: *Information Theory, IEEE Transactions on* 39.3 (May 1993), pp. 733–742. ISSN: 0018-9448. DOI: 10.1109/18.256484 (cit. on pp. 20, 21).

[67] Clive Maxfield. *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows.* Elsevier, 2004 (cit. on pp. 45, 46).

[68] D. Mayers. "Unconditional Security in Quantum Cryptography". In: *Journal of the ACM (JACM)* 48.3 (2001), pp. 351–406 (cit. on p. 18).

[69] Dominic Mayers. "Quantum Key Distribution and String Oblivious Transfer in Noisy Channels". In: *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings.* Vol. 1109. Lecture Notes in Computer Science. Springer, 1996, pp. 343–357. DOI: 10.1007/3-540-68697-5_26 (cit. on p. 18).

[70] D. A. McGrew and J. Viega. *The Galois/Counter Mode of Operation (GCM).* Submission to NIST Modes of Operation. 2005 (cit. on p. 39).

[71] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography.* 1st. Boca Raton, FL, USA: CRC Press, Inc., 1996. ISBN: 0849385237 (cit. on pp. 5, 13, 14).

[72] P.W. Milonni and M.L. Hardies. "Photons Cannot Always be Replicated". In: *Physics Letters A* 92.7 (1982), pp. 321–322. ISSN: 0375-9601. DOI: http://dx.doi.org/10.1016/0375-9601(82)90899-4 (cit. on p. 17).

[73] Michael Muehlberghuber et al. "FPGA-Based High-Speed Authenticated Encryption System". English. In: *VLSI-SoC: From Algorithms to Circuits and System-on-Chip Design.* Ed. by Andreas Burg et al. Vol. 418. IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg, 2013, pp. 1–20. ISBN: 978-3-642-45072-3. DOI: 10.1007/978-3-642-45073-0_1 (cit. on p. 2).

[74] M. Muehlberghuber et al. "100 Gbit/s Authenticated Encryption Based on Quantum Key Distribution". In: *VLSI and System-on-Chip (VLSI-SoC), 2012 IEEE/IFIP 20th International Conference on.* Oct. 2012, pp. 123–128. DOI: 10.1109/VLSI-SoC.2012.6379017 (cit. on p. 2).

[75] National Institute of Standards and Technology (NIST). *FIPS PUB 198-1, The Keyed-Hash Message Authentication Code (HMAC)*. Federal Information Processing Standards Publication 198-1, U.S. Department of Commerce. Available online: `http://www.itl.nist.gov/fipspubs`. July 2008 (cit. on pp. 14, 32).

[76] National Institute of Standards and Technology (NIST). *Special Publication 800-38A 2001 ED, Recommendation for Block Cipher Modes of Operation - Methods and Techniques*. Available online at `http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`. Dec. 2001 (cit. on pp. 14, 32).

[77] National Institute of Standards and Technology (NIST). *Special Publication 800-38C 2004, Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality*. Available online: `http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C.pdf`. May 2004 (cit. on pp. 32, 33, 35, 65).

[78] National Institute of Standards and Technology (NIST). *Special Publication 800-38D, Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. Available online: `http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf`. Nov. 2007 (cit. on pp. 3, 32, 33, 36, 39).

[79] James Nechvatal et al. "Report on the Development of the Advanced Encryption Standard (AES)". In: *Journal of Research of the National Institute of Standards and Technology* 106.3 (2001) (cit. on p. 23).

[80] NIST. *Advanced Encryption Standard (AES) (FIPS PUB 197)*. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`. 2001 (cit. on pp. 23–25, 27).

[81] NIST. *Cryptographic Hash Algorithm Competition Website*. Online: `http://csrc.nist.gov/groups/ST/hash/sha-3` (cit. on p. 65).

[82] NIST. *DRAFT FIPS PUB 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Available online: `http://csrc.nist.gov/publications/drafts/fips-202/fips_202_draft.pdf`. May 2014 (cit. on pp. 14, 65).

[83] NIST VCAT. *Cryptographic Standards and Guidelines Development Process: Report and Recommendations of the Visiting Committee on Advanced Technology of the National Institute of Standards and Technology*. Tech. rep. National Institute of Standards and Technology, July 2014 (cit. on pp. 3, 32).

[84] Asher Peres. "How to Differentiate Between Non-Orthogonal States". In: *Physics Letters A* 128.1 (1988), p. 19 (cit. on p. 21).

[85] *QCRYPT: Secure High-Speed Communication based on Quantum Key Distribution*. `http://www.nano-tera.ch/nanoterawiki/QCRYPT` (cit. on pp. 1, 48).

[86] Tang Qiong and Ye Jianwu. "Implementation and Optimization of AES Hardcore with High Performance Based on Bram". In: *Computer Science and Information Processing (CSIP), 2012 International Conference on*. Aug. 2012, pp. 699–702. DOI: `10.1109/CSIP.2012.6308950` (cit. on p. 2).

[87] Shanxin Qu et al. "High Throughput, Pipelined Implementation of AES on FPGA". In: *Information Engineering and Electronic Commerce, 2009. IEEC '09. International Symposium on*. May 2009, pp. 542–545. DOI: `10.1109/IEEC.2009.120` (cit. on p. 97).

[88] ID Quantique. *White Paper: Random Number Generation using Quantum Physics.* Online: `http://www.idquantique.com/images/stories/PDF/quantis-random-generator/quantis-whitepaper.pdf`. 2010 (cit. on p. 49).

[89] R Renner. "Security of Quantum Key Distribution". Presented on Sep 2005. PhD thesis. Zürich: Swiss Federal Inst. Technol., Zürich, 2005 (cit. on pp. 19, 21).

[90] Ronald L. Rivest et al. *The RC6 Block Cipher.* English (cit. on pp. 2, 23).

[91] P. Rogaway and D. Wagner. *A Critique of CCM.* daw@cs.berkeley.edu 12156 received 13 Apr 2003. 2003 (cit. on p. 35).

[92] Phillip Rogaway. "Authenticated-Encryption with Associated-Data". In: *In Proc. 9th CCS.* ACM Press, 2002, pp. 98–107 (cit. on p. 32).

[93] Phillip Rogaway et al. "OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption". In: *ACM Conference on Computer and Communications Security.* 2001, pp. 196–205 (cit. on p. 39).

[94] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli. "Architecture of Field-Programmable Gate Arrays". In: *Proceedings of the IEEE* 81.7 (July 1993), pp. 1013–1029. ISSN: 0018-9219. DOI: `10.1109/5.231340` (cit. on p. 46).

[95] M. Salter and R. Housley. *Suite B Profile for Transport Layer Security (TLS).* RFC 6460 (Informational). Internet Engineering Task Force, Jan. 2012 (cit. on p. 3).

[96] A. Satoh, T. Sugawara, and T. Aoki. "High-Performance Hardware Architectures for Galois Counter Mode". In: *Computers, IEEE Transactions on* 58.7 (July 2009), pp. 917–930. ISSN: 0018-9340. DOI: `10.1109/TC.2008.217` (cit. on p. 53).

[97] Valerio Scarani et al. "The Security of Practical Quantum Key Distribution". In: *Rev. Mod. Phys.* 81 (3 Sept. 2009), pp. 1301–1350. DOI: `10.1103/RevModPhys.81.1301` (cit. on p. 19).

[98] Bruce Schneier et al. "Twofish: A 128-Bit Block Cipher". In: *in First Advanced Encryption Standard (AES) Conference.* 1998 (cit. on pp. 2, 23, 63).

[99] C. E. Shannon. "A Mathematical Theory of Communication". In: *SIGMOBILE Mob. Comput. Commun. Rev.* 5.1 (Jan. 2001), pp. 3–55. ISSN: 1559-1662. DOI: `10.1145/584091.584093` (cit. on pp. 6, 7).

[100] C.E. Shannon. "Communication Theory of Secrecy Systems". In: *Bell System Technical Journal, The* 28.4 (Oct. 1949), pp. 656–715. ISSN: 0005-8580. DOI: `10.1002/j.1538-7305.1949.tb00928.x` (cit. on pp. 7, 21).

[101] Peter W. Shor and John Preskill. "Simple Proof of Security of the BB84 Quantum Key Distribution Protocol". In: *Phys. Rev. Lett.* 85 (2 July 2000), pp. 441–444. DOI: `10.1103/PhysRevLett.85.441` (cit. on p. 18).

[102] Mostafa I. Soliman and Ghada Y. Abozaid. "FPGA Implementation and Performance Evaluation of a High Throughput Crypto Coprocessor". In: *Journal of Parallel and Distributed Computing* 71.8 (2011), pp. 1075–1084. ISSN: 0743-7315. DOI: `http://dx.doi.org/10.1016/j.jpdc.2011.04.006` (cit. on p. 2).

[103] Jaroslaw Sugier. "Implementing Serpent Cipher in Field Programmable Gate Arrays". In: *The 5th International Conference on Information Technology, ICIT.* 2011, pp. 11–13 (cit. on pp. 3, 63, 64, 94).

[104] *The Serpent Homepage.* URL: http://www.cl.cam.ac.uk/~rja14/serpent.html (cit. on p. 82).

[105] N. Walenta et al. "A Fast and Versatile Quantum Key Distribution System with Hardware Key Distillation and Wavelength Multiplexing". English. In: *NEW JOURNAL OF PHYSICS* 16.1 (2014), pp. 1–20 (cit. on p. 48).

[106] Jimei Wang et al. "High-Speed Architectures for GHASH Based on Efficient Bit-Parallel Multipliers". In: *Wireless Communications, Networking and Information Security (WCNIS), 2010 IEEE International Conference on.* 2010, pp. 582–586. DOI: 10.1109/WCINS.2010.5541846 (cit. on p. 4).

[107] Stephen Wiesner. "Conjugate Coding". In: *SIGACT News* 15.1 (Jan. 1983), pp. 78–88. ISSN: 0163-5700. DOI: 10.1145/1008908.1008920 (cit. on p. 15).

[108] W. K. Wootters and W. H. Zurek. "A Single Quantum Cannot Be Cloned". In: *Nature* 299.5886 (Oct. 28, 1982), pp. 802–803. DOI: 10.1038/299802a0 (cit. on p. 17).

[109] Hongjun Wu and Tao Huang. *The Authenticated Cipher MORUS (v1).* Available online: http://competitions.cr.yp.to/round1/morusv1.pdf. Mar. 2014 (cit. on pp. 3, 65).

[110] Hongjun Wu and Bart Preneel. "AEGIS: A Fast Authenticated Encryption Algorithm". English. In: *Selected Areas in Cryptography – SAC 2013.* Ed. by Tanja Lange, Kristin Lauter, and Petr Lisoněk. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 185–201. ISBN: 978-3-662-43413-0. DOI: 10.1007/978-3-662-43414-7_10 (cit. on p. 33).

[111] Hugo Zbinden et al. "Continuous QKD and High Speed Data Encryption". In: *Proc. SPIE 8899, Emerging Technologies in Security and Defence; and Quantum Security II; and Unmanned Sensor Systems X.* Vol. 8899. 0P. 2013, pp. 1–4. DOI: 10.1117/12.2032731 (cit. on p. 2).

[112] Bob Zeidman. *Designing with FPGAs and CPLDs.* CRC Press, 2002 (cit. on pp. 45, 46).

[113] Gang Zhou, Harald Michalik, and László Hinsenkamp. "Improving Throughput of AES-GCM with Pipelined Karatsuba Multipliers on FPGAs". In: *Reconfigurable Computing: Architectures, Tools and Applications.* Ed. by Jürgen Becker et al. Vol. 5453. Lecture Notes in Computer Science. 10.1007/978-3-642-00641-8_20. Springer Berlin / Heidelberg, 2009, pp. 193–203. ISBN: 978-3-642-00640-1 (cit. on pp. 3, 54, 94).