TU UB

Die approbierte Originalversion dieser Diplom-/ Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.

http://www.ub.tuwien.ac.at

TU UB WIEN Universitätsbibliothek

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.

http://www.ub.tuwien.ac.at/eng

DIPLOMA THESIS

Hardware Acceleration of Cryptographic Procedures for Secure Distributed Storage Systems

Submitted at the Faculty of Electrical Engineering and Information Technology,
Vienna University of Technology
in partial fulfillment of the requirements for the degree of
Diplom-Ingenieur (equals Master of Sciences)

under supervision of

Univ.Prof. Dipl.-Ing. Dr.techn.Axel Jantsch
Univ.Ass. Dr. Sai Manoj Pudukotai Dinakarrao

Institute of Computertechnik Technology (E384)
Vienna University of Technology

by

Jakob Stangl
Matr.Nr. 0925320
Schellenseegasse 11/5, 1230 Wien

25.04.2017

Abstract

Cloud computing gives new opportunities to many applications. The advantages of reducing the cost and complexity of up-front infrastructure are accompanied with various security issues.

A cloud storage provider has to be trusted to handle ones data confidentially and ensure its availability. In order to guarantee the secrecy and availability of sensitive data, it is unavoidable for the end-user to encrypt and replicate the data in any way. An approach for these issues is to store the data in a virtual cloud formed of multiple cloud providers.

With a scheme called secret sharing a free selectable amount of redundancy is added. The data to be protected is split into multiple pieces and distributed to several servers. To restore the original data a certain amount of arbitrary pieces is required. In this way, the failure of a single server in the cloud does not influence the availability. Moreover, this schemes enables high secrecy as only a certain amount of pieces reveal the original information. Any party holding less pieces are not capable of obtaining any information of the original data.

It is of high computational effort to split the data in such a manner. While various software solutions exist, there are only few investigations in hardware. However, the implantation in dedicated hardware allows the possibility of high performance increase and has the potential to expand its applicability.

This work firstly handles the implementation of an information theoretical secure secret sharing scheme, proposed by Adi Shamir. Subsequently, it discusses and presents the implementation of a more efficient scheme in terms of storage space, the Computational Secret Sharing. All these investigations are targeted for a Field Programmable Gate Array (FPGA). A final implementation of a complete secret sharing system operating in a network environment and the capability of managing, sharing and distributing complete files as well as successfully restoring them completes this work.

Kurzfassung

Viele Anwendungen wurden durch das Zusammenschließen von Computern zu einer *Cloud* erst möglich oder bedeutsam verbessert. Allerdings ergeben sich durch die Auslagerung von Daten auch neue Bedenken. Speichert man etwa Daten in der *Cloud*, so muss man dem Anbieter vollkommen vertrauen, die Daten zu schützen, sowie auch ihre Verfügbarkeit sicherzustellen. Letztendlich liegt es jedoch in der Verantwortung des Nutzers vertrauliche Daten entsprechend zu schützen und sich gegen den Ausfall der *Cloud* oder Datenverlust abzusichern. Dies führt unweigerlich zu Verschlüsselung und Replikation.

Eine Lösung dies zu erreichen besteht darin, eine virtuelle *Cloud* aufzubauen, die aus mehreren physischen *Clouds* besteht. Mittels eines Verfahrens namens *SecretSharing* wird zu den Daten eine beliebige Menge an Redundanz hinzugefügt. Dabei werden die Originaldaten in mehrere Teile zerlegt und in der virtuellen *Cloud* auf unterschiedliche Server verteilt. Um die Originaldaten wiederherzustellen, wird eine vordefinierte Anzahl dieser Teile rekombiniert. Dabei ist es gleichwertig, welche dieser Teile für die Rekombination verwendet werden. Dank dieser Methode wird der Ausfall einer definierbaren Anzahl von Servern kompensiert. Zusätzlich wird mit diesem Verfahren Vertraulichkeit sichergestellt. Mit weniger als für die Wiederherstellung benötigten Teile lässt sich keine Information über den Inhalt der Daten ableiten.

Für dieses Konzept existiert eine Vielzahl an Softwarelösungen, jedoch nur eine sehr limitierte Auswahl an Hardwarerealisierungen. Eine eigens dafür konzipierte Hardware in dieser Arbeit bringt erhebliche Durchsatzsteigerungen und erweitert somit den Anwendungsbereich bei gesenkten Kosten.

Zunächst wird in dieser Arbeit ein informations-theoretisch sicheres Verfahren, erstmals vorgestellt von Adi Shamir, in Hardware implementiert. Darauf aufbauend wird das *Computational Secret Sharing*, welches eine bessere Speichereffizienz besitzt, untersucht. Für beide Ansätze werden generische Hardwarearchitekturen für die Realisierung in einem FPGA entworfen. Abschluss dieser Arbeit ist die Implementierung eines kompletten Systems in einer Netzwerkumgebung, die in der Lage ist mehrere Dateien zu verwalten, zu teilen und wiederherzustellen.

Danksagung

An erster Stelle möchte ich meinen Eltern Renate und Gerhard Stangl danken, ohne die ich heute diese Arbeit nicht hätte schreiben können. Ich möchte mich bedanken, dass sie mir nicht nur ein sorgenfreies Studieren ermöglicht haben, sondern mich überdies dazu ermutigt haben ein universitäres Studium abzuschließen.

Für die intensive und kompetente Betreuung von Thomas Lorünser von seitens des Austrian Institute of Technology (AIT) möchte ich mich bedanken. Seine fachlichen Anregungen und Analysen waren von großer Hilfe für diese Arbeit.

Für die Betreuung meiner Arbeit seitens der TU Wien, möchte ich mich bei Axel Jantsch bedanken, der es mir ermöglicht hat die Diplomarbeit mit hoher fachlicher Unterstützung in einem angenehmen Umfeld zu schreiben.

Bei Sai Manoj Pudukotai Dinakarrao möchte ich mich ebenfalls für die kompetente und unkomplizierte Betreuung seitens der TU Wien bedanken. Seine flexible und spontane Art waren mir bei vielen Problemen von großer Hilfe.

Mein ganz besonderer Dank in Bezug auf diese Arbeit gilt Jasmin Pajenda. Ohne ihre unermüdliche Bereitschaft neue Ideen zu diskutieren und den langen konstruktiven Gesprächen wäre diese Arbeit nicht in diesem Ausmaß gelungen.

Table of Contents

1 Introduction **1**
- 1.1 Secret Sharing In The Cloud . . . 1
- 1.2 Motivation For A Hardware Implementation . . . 2
- 1.3 Objectives . . . 3
- 1.4 About This Work . . . 3
- 1.5 Organisation Of Thesis . . . 3

2 Preliminaries And Related Work **4**
- 2.1 Finite Fields . . . 4
- 2.2 Secret Sharing Principles . . . 5
- 2.3 Shamirs Secret Sharing . . . 5
- 2.4 Computational Secret Sharing . . . 7
- 2.5 Robust And Verifiable Secret Sharing . . . 7
- 2.6 Hardware Implementations . . . 9
- 2.7 Software implementations . . . 9

3 Shamir's Secret Sharing **11**
- 3.1 Introduction . . . 11
- 3.2 Share Generation . . . 11
- 3.3 Secret Reconstruction . . . 18
- 3.4 Evaluation . . . 20

4 Advanced Multipliers **25**
- 4.1 Introduction . . . 25
- 4.2 Applying Karatsuba's Algorithm . . . 25
- 4.3 A Better Overall Resource Utilization . . . 28
- 4.4 Conclusion . . . 32

5 Computational Secret Sharing **34**
- 5.1 Introduction . . . 34
- 5.2 Advanced Encryption Standard . . . 35
- 5.3 Share Generation . . . 39
- 5.4 Secret Reconstruction . . . 39
- 5.5 A Full Computational Secret Sharing Core . . . 41
- 5.6 Evaluation . . . 43

6 A Full Computational Secret Sharing System 47
6.1 Introduction 47
6.2 True Random Number Generator 48
6.3 Protocol 50
6.4 External Communication 51
6.5 Computational Secret Sharing Core Wrapper 52
6.6 Client And Server 53
6.7 Full Setup And Evaluation 54

7 Conclusion 57
7.1 Summary And Discussion 57
7.2 Applications 58
7.3 Further Work 58

Appendix 59

Literature 86

Abbreviations

ARP	Address Resolution Protocol
AES	Advanced Encryption Standard
ADC	Analog to Digital Converter
AIT	Austrian Institute of Technology
AXI	Advanced Expandable Interface
BRAM	Block Random Access Memory
CBC	Cipher Block Chaining
CFB	Cipher Feedback
clk	Clock
CSP	Cloud Service Provider
CSS	Computational Secret Sharing
CTR	Counter
CRC	Cyclic Redundancy Check
D-FF	D-Flip-Flop
DSP	Digital Signal Processor
ENC	Encryption Function
ECB	Electronic Code Block
ESS	Extended (Shamir) Secret Sharing
FFT	Fast Fourier Transformation
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FIFO	First In- First Out
GF	Galois Field
GUI	Graphical User Interface
RGMII	Gigabit Media Independent Interface
GPU	Graphics Processing Unit
HW	Hardware
IP_1	Intellectual Property
IP_2	Internet Protocol
ISO	International Organisation for Standardisation
LFSR	Linear Feedback Shift Register
LUT	Look-Up-Table
MAC	Media Access
NIST	National Institute of Standards and Technology
NSA	National Security Agency
OLED	Organic Light-Emitting Diode
OFB	Output Feedback
PEU	Polynomial Evaluation Unit
PLL	Phase Locked Loop
PL	Programmable Logic

PS	Processing System
PEU	Polynomial Evaluation Unit
RAID	Redundant Array of Independent Disk
RO	Ring Oscillator
SRU	Secret Reconstruction Unit
SHA	Secure Hash Algorithm
SLA	Service-Level Agreement
SSS	Shamir Secret Sharing
SGU	Share Generation Unit
SW	Software
TDP	Thermal Design Power
TRNG	True Random Number Generator
VHDL	Very High Speed Integrated Circuit Hardware Description Language
WPA2	Wi-Fi Protected Access 2

1 Introduction

This work starts with an outline of computational clouds and the application fields within the cloud setup. While various software (SW) solutions exist, a special motivation for a hardware (HW) acceleration is given. The general conditions for implementations is listed as well as the outline and notations of this work at the end of the chapter.

1.1 Secret Sharing In The Cloud

In the field of information technology the purpose of a cloud is outsourcing resources and reducing the complexity and cost of up-front infrastructure, rapidly realizable with minimal management effort. While cloud computing has a wide field of application, the focus of this work is on data storage. However, handing the data to a Cloud Service Provider (CSP) of a cloud leads to various issues. The costumer has to trust the service provider to handle the data confidentially. This is of special interest for companies or individuals, if the data contains sensitive information. Moreover, the data should be available at any time when requested.

(A a minimum uptime for cloud services is specified in the service-level agreement (SLA), which is ..) In the service-level agreement (SLA), a minimum uptime is specified for cloud services. For example the popular Amazon Cloud Service the SLA value is 99.9% uptime per month [**?**]. Anyway, as shown by recent incidents in [15] a downtime greater than the specified value is still possible, but only allows to receive redress. It is still in the responsibility of the user to ensure compliance of the final security and confidentiality [8].

Various different approaches to overcome those problems include replicating the data and encryption. Another attempt is the cloud-of-cloud approach. In this scenario a virtual cloud consisting of more than one physical cloud provider is in charge and realized by a process called secret sharing. Various systems exist and some of them are summarized in [67].

In the secret sharing approach, a file is split into pieces (shares) and stored on multiple servers within different clouds. In order to obtain the original file, an arbitrary subset of these shares are required. In this way, the reliability of a single cloud or server is avoided, if sufficient shares are generated. For example, if 10 servers are in use with an uptime of 99.9 % with 4 required shares, the failure probability is brought from 10^{-3} to 1.1×10^{-16} [49].

Moreover, it is of special advantage, that a certain amount of shares, k, is required to obtain the original file. With less shares than this boundary, no information is obtained and of no use

for a malicious cloud provider, or attacker. It ensures confidentiality if less than k share holders collaborate.

Compared to the straight forward approach of encryption and duplication of files, the scheme of this work allows two advantages. Firstly, no key management is required as no key exists in the scheme and secondly, the data overhead is adjustable. Due to possible malicious attacks, there are still credentials required to authenticate the shares and guarantee a consistent and secure secret-file. However, such an authentication mechanism is required anyway in an encrypted scheme.

1.2 Motivation For A Hardware Implementation

To realize the concept of secret sharing a mathematical algorithm is applied subsequently. Performing the algorithm on a file, it is first split into blocks of a certain bit-width. The block can be seen as numbers on which the algorithm is applied. On one hand, blocks of higher bit-widths lead to a mathematical higher complexity due to multiplications in the algorithm. On the other hand, a higher bit-width comes with benefits. Even in a computational secure scheme, it is possible to guess a right secret, but every possibility is same likely. Using higher bit-widths makes it significantly more unlikely to guess a right word. Moreover, the performance of higher protocols in a cloud-of-cloud approach is increased. The protocol [21] for auditing [9] and [39], a process to verify the consistency of shares without retrieving the original file benefits of bigger blocks. Furthermore, the scheme of secret sharing is homomorph. Consider two subsecrets and each split into multiple shares. Multiplying shares from both subsecrets lead to a new topsecret. This topsecret is the same as both subsecrets multiplied. In this case, a topsecret is generated, without revealing any subsecrets. However, this homomorphic attribute is only applicable within a block.

Because of the performance penalties in the storage domain software implementations usually perform the calculations on blocks of one byte, as the calculations are able to be performed efficiently in look-up-tables. For processing 16 bits at once, other look-up-table based approaches exit, but the calculation becomes significantly slower for higher bit-widths and inefficient in terms of possible throughput. Therefore current software implementations of secret sharing, such as proposed in Archistar [49] work with a block size of 8 bit and can achieve a share generation rate of about 32 Mbit/s. Another implementation of Nubisave in [68] reach a throughput of about 5 Mbit/s in a real world environment. In chapter 2.7 more software implementations of secret sharing algorithms are discussed.

For the improvement of the performances in terms of their throughput and bit-widths, a hardware implementation is of interest. Especially when we want to apply the technology within a larger data centre which also requires low latency. According to the block size, the calculations are performed on a hardware designed for a certain bit-width.

To the best of my knowledge, only two hardware implementations of secret sharing are published, [77] and [50]. Gaining a high throughput in both works, the sharing and reconstruction process as well as the evaluation of the architecture for different bit-widths was not the intention in these works.

Therefore, this work discusses the feasibility of a hardware implementation and evaluates the performances of different bit-widths. It is focused on different secret sharing algorithms and word widths, specifically optimized for storage configuration motivated by the use cases covered by the ARCHISTAR system [49].

1.3 Objectives

The aim of this thesis is to evaluate the feasibility to realize a secret sharing mechanism in a FPGA for different bit-widths specifically targeted for efficient distributed storage solutions. Hardware units are designed and bit-widths are discussed, which are generically selectable. In the following, the bit-widths 8, 16, 32, 64 and 128 were investigated and implications for practical realizations have been elaborated.

The work firstly deals with a secret sharing scheme of information theoretical secrecy, the Shamir Secret Sharing (SSS) scheme [65]. It forms the foundation for a more complex secret sharing scheme with better storage efficiency, the Computational Secret Sharing (CSS) scheme [46], investigated subsequently.

After evaluation of the performance in different bit-widths, a prototype is built. It is capable of sharing and restoring files with a high throughput in a real network environment.

1.4 About This Work

Evaluation Platform

The Zedboard [82] was selected as FPGA platform. The hardware description code was developed in Very High Speed Integrated Circuit Hardware Description Language (VHDL). The synthesis and implementations of this work were performed with Vivado® 2015.3 [79] and targeted for the Zedboard. The Zedboard consists of a Xilinx Zynq-7000 AP SoC XC7Z020-CLG484 [80] FPGA. It is partitioned into a Programmable System (PS) and a Programmable Logic (PL), connected via an Advanced Expandable Interface (AXI). The PS contains a dual-core ARM Cortex™-A9 processor. The PL consists of 6-input look-up-tables (LUTs), D-Flip-Flops (D-FFs), 36 kbit Block Random Access Memories (BRAMs) and Digital Signal Processors (DSPs) as well as several interfaces.

Notations

In figures presenting architectures of hardware designs, bold lines symbolize buses (usually data paths), while thin lines represent single signals (usually control signals). The abbreviation LUT refers to the logic element within a FPGA, while written out refers to a look-up-table in general terms.

1.5 Organisation Of Thesis

The chapter 2 describes fundamental basics, important for the understanding of this work as well as related work. Then, the implementation of Shamir's Secret Sharing scheme, an information theoretical secure secret sharing scheme is investigated in 3. The next chapter 4 contains strategies for more efficient implementations of a polynomial multiplier. In chapter 5 the development from the implementation of Shamir's scheme to a Computational Secret Sharing scheme is presented. A complete prototype capable of sharing files in a network environment is presented in chapter 6. Finally, a conclusion and outlook of this work is in 7. All chapters with implementations and findings include a dedicated evaluation and summary subsection.

2 Preliminaries And Related Work

In this chapter an introduction of the main elements and important topics, such as the detailed description of Shamir's Scheme and finite fields, is given. Starting with finite fields, new concepts in each section are introduced in order to get a complete understanding of a robust computational secret sharing scheme. The following sections provide an overview of related work, which will be the content of the rest of this chapter.

2.1 Finite Fields

In a secret sharing algorithm a lot of additions and multiplications have to be performed sequentially on an initial value. If the operands are represented by natural numbers, it leads to a rapid increase of the original value. A large number of bits would be needed to represent the highest possible value in a digital system to avoid overflows. However, most of the bits are often not used which is very inefficient. For an efficient use of these bits all calculations are performed in finite fields, or more specific, in binary extension fields.

For detailed information of finite fields and related mathematical basics it is referred to [48] while a brief introduction is given in the following.

Finite fields, denoted as $\mathbb{F}_p$, consist of a finite number p of elements and satisfy all rules of fields. They are often denoted as a Galois field, GF(p), in honour of Évariste Galois. The simplest construction of finite fields uses prime numbers for p. Operations of arithmetic, such as addition and multiplication of any operand within this field, lead to a result in the same field. This is necessary to satisfy the axioms of fields and is only possible if p is a prime. To obtain such a result, a reduction step (modulo p) is performed after each operation. While the numbers don't grow larger, each operation can still be unambiguously reversed. In order to gain a better understanding a simple example for $\mathbb{F}_7$ is given in (2.1).

$$\begin{aligned} 3 + 5 &= 8 \bmod 7 = 1 \\ 1 - 5 &= -4 \bmod 7 = 3 \end{aligned} \tag{2.1}$$

Extension fields, denoted as GF(p^n), are used for constructing fields with a number of elements different to prime numbers. These fields consist of p^n elements, where p stands for a prime and n any natural number. In order to construct such a field, the elements are represented by polynomials with a degree less than n and coefficients are described by elements of the field $\mathbb{F}_p$.

Similar to the first example with finite fields of p elements, a reduction is performed after each operation to stay in the field. Instead of the prime number an irreducible polynomial is used, which is chosen arbitrary but unique for a specific binary extension field. The reduced polynomial is the remainder of a division by the irreducible polynomial. A simple example of a multiplication in GF(2^2) with the irreducible polynomial $x^2 + x$ is given in 2.2.

$$(x+1)(x+0) = (x^2 + x) \bmod (x^2 + x + 1) = 1 \tag{2.2}$$

If p is chosen as 2, the coefficients can only be 0 or 1. Such fields are named binary extension fields and often denoted as GF(2^n), where n signifies the bit-widths. They can be applied easily and efficiently in digital systems.

2.2 Secret Sharing Principles

Secret sharing is an algorithm for distributing one secret into several share parts among different parties. The secret is divided into shares, where a specific number of shares is needed to restore the secret. In a n/k threshold scheme n shares are produced, while a minimum of freely chosen k shares are needed to reconstruct the secret. Any number of shares less than k does not reveal any information about the secret.

An algorithm to meet this criteria was first independently introduced by Adi Shamir [65] and George Blakley [10]. Shamir's Secret Sharing algorithm is based on two-dimensional polynomials, while Blakley's algorithm works with planes and their intersection in a n-dimensional space. Shamir's algorithm is more efficient in a storage setting, because it produces smaller overheads. A share of Blakley's algorithm is the size of n times the secret, whereas one share in Shamir's model has exactly the size of the secret. Shamir size is optimal for information theoretical security, however, for practical reasons it can be more efficient with slightly weaker security assumptions. Therefore the basic algorithm was adapted to a more advanced secret sharing mechanism to meet more requirements such as smaller overheads, robustness and verification.

2.3 Shamirs Secret Sharing

Shamir's Secret Sharing is based on the evaluation and interpolation of polynomials. For a n/k threshold scheme a polynomial with a degree of k-1 is used (2.3). The first coefficient m at x^0 represents the secret, while all other coefficients, $c_x ... c_1$, are filled with random numbers. To construct a share the polynomial is evaluated at a point x, where x can be chosen arbitraryily except $x \neq 0$, which is the secret itself.

$$\begin{aligned} y = c_{k-1} \times x^{k-1} + ... + c_2 \times x^2 + c_1 \times x^1 + c_0 \\ \text{where:} \quad y \quad &= \quad \text{share} \\ c_0 \quad &= \quad \text{secret} \\ c_1 ... c_{k-1} \quad &= \quad \text{random values} \end{aligned} \tag{2.3}$$

A polynomial interpolation is performed to reconstruct the secret. Any share represents a point of the polynomial, where the x-position is publicly known and the y-position is the share itself.

The y-value at $x = 0$, or equally the coefficient at x^0 from the interpolated polynomial, is the secret. For an exact interpolation of a polynomial with a degree of k-1, a minimum of k points is required. If only k-1 points are available it is obvious, that the polynomial can't be reconstructed as the secret could still take any value depending on the last share-point.

In this scheme the size of the shares are exactly the size of the secret. Therefore an overhead of n-times the secret is produced. An advantage of this scheme is its information-theoretic secrecy, which means it can not be broken with any computing effort.

For a deeper understanding of Shamir's Secret Sharing an illustration is given in figure 2.1.

We choose a polynomial with a degree of 2 - $c_2 * x^2 + c_1 * x^1 + c_0$. c_0 represents the secret, which we assume is 4. The coefficients c_1 and c_2 are randomly chosen with 2 and -5. Next, we calculate 4 shares $s_1 ... s_4$ on the points 1...4, which gives us $s_1 = 1, s_2 = 2, s_3 = 7, s_4 = 16$.

We select the shares s_1, s_2, s_4 and use the Lagrange Interpolation at the point $x = 0$ (2.4). There are many other interpolation schemes which could be used equally.

$$\sum_{i=0}^{n} \left(f(i) * \prod_{j=0, j \neq i}^{n} \frac{0 - x_j}{x_i - x_j} \right) \tag{2.4}$$

The Lagrange Interpolation leads to the following equation (2.5) with the result of the original secret we selected.

$$y = \left(1 * \frac{-2}{1-2} * \frac{-4}{1-4}\right) + \left(2 * \frac{-1}{2-1} * \frac{-4}{2-4}\right) + \left(16 * \frac{-1}{4-1} * \frac{-2}{4-2}\right) = 4 \tag{2.5}$$

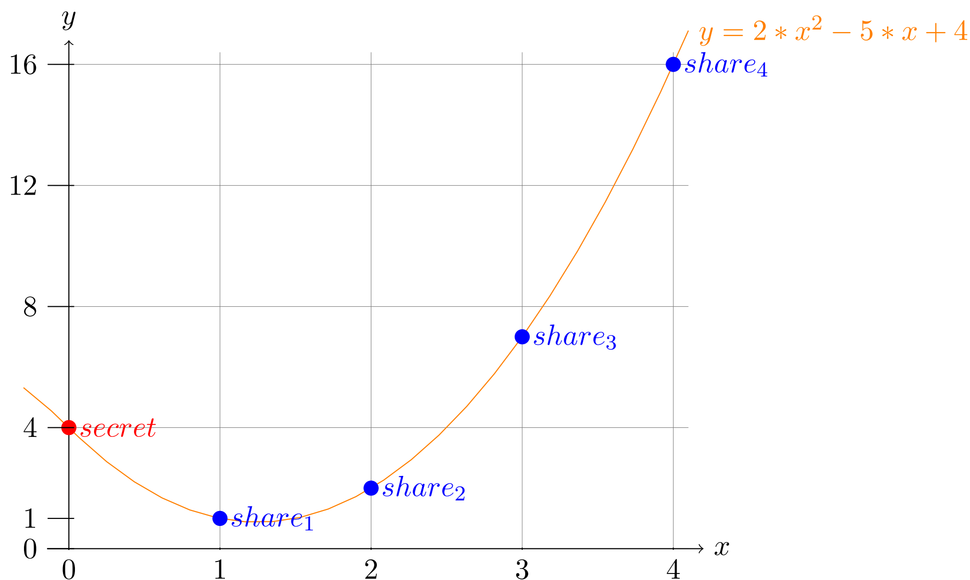


Figure 2.1: An illustration of a polynomial for Shamir's Secret Sharing of a 4/3 threshold scheme, using 4 as the secret and $1, 2, 7, 16$ as random coefficients.

2.4 Computational Secret Sharing

The information-theoretic secrecy comes with the cost of shares with sizes of the secret itself. Although in practical implementations it is often sufficient if the secret is computational secure. The secret can not be revealed with a limited computational effort. Hugo Krawczyk published such a scheme called Computational Secret Sharing [46].

Computational Secret Sharing is described with an abstract secure secret sharing scheme. For a better understanding a brief explanation is given based on Shamir's scheme. Instead of random numbers it uses secrets in higher coefficients of the polynomial and a secure encryption function.

First the secret is encrypted with a secure encryption function (ENC) using a key A. The shares $s_1, ..., s_i$ are generated by evaluating a polynomial at any point $x_i \neq 0$ and each coefficient is part of the secret. This is the fundamental difference to Shamirs's scheme, where all coefficients except c_0 are filled with random numbers. The key is then shared with a perfect secure secret sharing scheme into key-shares K_i. Each pair, consisting of a secret-share s_i and a key-share K_i, is distributed to one shareholder.

In the process of reconstructing the secret, first the key is restored followed by the encrypted secret. In comparison to Shamir's scheme all coefficients of the polynomial are needed to be restored. The result is then decrypted by using the restored key and the reversed encryption function ENC^{-1} to reveal the original secret.

2.5 Robust And Verifiable Secret Sharing

Computational Secret Sharing makes the scheme efficient and computational secure, but the scheme is still vulnerable for attacks. If a shareholder distributes a modified share, it is not feasible for the system to detect the faulty share and a wrong secret would be the result. The error can not be detected nor can the faulty share be identified.

Robust Secret Sharing

A secret sharing system is called robust, if such a malicious share is detected and the original secret can still be recovered. To be more specific, it is (t, δ)-robust, where t is the maximal number of corrupted shares.

Shamir's scheme is robust for the case of $t < n/3$, $t < k$ and if all shares are available. In this case all possible combinations of shares could be reconstructed and a majority vote would show the correct secret. However, this scheme works to an upper bound of $n/3$ and is not efficient in term of computational effort, because a lot of reconstructions would be needed.

R.J McEliece and D.V. Sarwate first published a solution for this problem [53] by using error correcting codes. This scheme is also bounded by $t < n/3$. Another solution was presented by Tompa and Woll [70], where they use a much larger space than the secret itself. Therefore the probability for a recovered secret to be outside the original possible space is high, if at least one of its shares were corrupted. However, there is still a probability greater zero that the fault might not be detected and the scheme is inefficient in terms of space usage.

Rabin and Ben-Or used message authentication codes [57]. Their scheme is often refereed to as information checking. Every share s_i gets extended by an authentication tag $\tau_{i,j}$ for every shareholder P_j and the key is then send to the corresponding P_j. At the secret reconstruction it is checked if the share is accepted my sufficient many authentication tags.

In [56] Rabin introduced a scheme based on digital signatures. This was extended by H. Krawczyk in [46] to overcome the problem of key-management and increase the space efficiency. It is based on distributed fingerprints [45] and published by Krawczyk. The fingerprints are generated with hash-functions and divided into shares, for example with schemes described above. These shares are now distributed to the shareholders. Using hashes as here is only suitable for computational secret sharing, because the hashes leak information. Contrary, the information checking above is information theoretical secure. In the following the share construction applying distributed fingerprints is outlined.

1. Calculate n shares s_i out of the secret
2. Calculate fingerprint of each share f_j with a Hash function
3. Calculate shares of all fingerprints s_{fj}
4. Every shareholder gets one share of the secret s_i and one share from each fingerprint s_{fj} $(s_i + s_{f0} + ... + s_{fj})$

Verifiable Secret Sharing

In the scenarios of robust secret sharing we assume that the dealer, who distributes the shares to the shareholders, is honest. If the dealer is corrupt he can distribute modified shares, the restored secret would depend on the shareholders selected for reconstruction. A scheme to detect such a behavior is called Verifiable Secret Sharing.

The first scheme to fulfil this requirement was published by Chor et. al [14]. There are a couple more solutions which are working for a limited amount of corrupted nodes (e.g. $t < n/3$) and unconditionally private, e.g. [34], [35] or with the assumption of a limited computational power of the attacker. Such computational secure schemes schemes can work with a higher boundary of corrupted nodes.

Auditing

The process where shares are verified if they are still valid is called Auditing. It is of benefit if this can be done without downloading all information and without reconstructing the secret itself. This gives convenience and the possibility to outsource this task to a third party which may be untrusted. Such schemes for the proof of retrieve-ability have been first introduced by Ateniese et al. [9] and Jules and Kalinski [39].

In the meantime a lot of schemes for single server setups e.g. [64] [63] [12] as well as for cloud of cloud setups, e.g. [21], have been proposed.

However, specifically for secret shared data a new protocol has been proposed recently [21] which leverages the redundancy in systems with $k < n$ and can be very efficiently applied to large sets of stored data, e.g., they introduce almost no overhead for practical configurations.

2.6 Hardware Implementations

There are only few hardware implementations of secret sharing algorithms known from the literature and no extensive treatment of such has been done so far. The focus in these works is different from the one in this thesis and to the best of knowledge there is no speed optimized FPGA implementation of the full Computational Secret Sharing scheme. Moreover, no work deals with the evaluation of different bit-widths of the secret to be processed at once.

In [77] secret sharing is used in a network monitoring application. The monitored data is encrypted and the key is shared using Shamir's algorithm. Only if enough evidence is collected the specific key is available to a monitoring entity. Therefore the encryption and share generation must be able to work for Gigabit-networks. To meet this criteria a dedicated hardware implementation in a FPGA was developed. Because the share generation is time critical, only the implementation of this part is described, while improvements were given also on the reconstruction algorithm.

The key is derived using an Advanced Encryption Standard (AES) function, a master key and the packet flow identifier. This key is used for the coefficient c_0 in the Shamir polynomial which is to be shared. All other coefficients in the Shamir polynomial are also generated over the same AES method. The generated key is then used to encrypt the payload of the package with an other AES unit, while the coefficients are used in a Shamir core. Beside the coefficients, the core also gets a x-value as input, which is supplied by a True Random Number Generator (TRNG).

For the evaluation of the Shamir polynomial, the Horner-Scheme is applied. This gives the possibility to break $m-1$ squarings and multiplications down to $m-1$ multiplications, where $m-1$ is the degree of the polynomial. These multiplications are then processed sequentially using the same multiplier. After each multiplication an addition and a reduction step follows. The reduction step is necessary because the algorithm works in a binary extension field. Since a fixed irreducible polynomial was used, the reduction can be done in one step with static XOR-connection. The x-coordinate was restricted to a 32 bit value in order to reduce the size of the multiplier. A 191×16 bit multiplier was used, which processes the 191×32 bit multiplication in two cycles.

The work was implemented on a Network FPGA card using a Virtex-II Pro 50 FPGA. The isolated examination of the share generation reveals a throughput of 2359Mbit/s with the usage of 1633 slices in this design. This are roughly 3266 4-Input look-up-tables. The full keyshare units reach a throughput of 343 Mbit/s with 3687 slices and 18 instances of BRAMs.

Another implementation from [50] focuses on secure secret sharing. In comparison to the implementation above, it's target architecture are ASICs and therefore it was synthesized with Cadence Encounter RTL Compiler with the Nangate 45nm Opencell library. While the Secret Sharing is an effective way to divide a secret, it is vulnerable to attackers and cheaters as mentioned in section 2.5. In their work they apply robust codes and algebraic manipulation detection to resist strong cheating attacks. Besides the size of the implementation the results are focusing on the efficiency of cheating detection and correction and the causing size-overhead.

2.7 Software implementations

There are various software implementations on secret sharing. The crucial parts of software implementation are the time-consuming polynomial multiplications limiting the performance.

Multiplications of small word-widths can be processed efficiently with look-up-tables, but the sizes growth exponentially with the used Galois field and respectively the bit-widths. For example, an 8×8 bit multiplication for a GF(2^8) would require $2^{8+8}/2 = 32768$ entries of 1 Byte. The size of the look-up-table would be about 32kByte and easily realizable in a common computer system. However, a look-up-table in GF(2^{16}) would need 4.3TB which is currently not feasible in an efficient manner.

The implementation of GF(2^{16}) can be efficiently done with logarithms and logarithm look-up-tables. Therefore the problem is transformed according to equation (2.6), where a and b are the factors and c the result of the multiplication.

$$a * b = c = log^{-1}(log(a * b)) = log^{-1}(log(a) + log(b)) \tag{2.6}$$

The logarithm and inverse logarithm can be stored in a look-up-table with 2^{16} entries and only the addition has to be performed.

However, after a certain size of the Galois field the method can not be applied anymore and the plain multiplication has to be performed by shift, AND and XOR operations.

In [42] a collection of libraries supporting secret sharing where gathered. The GFShare library [24] operates in a GF(2^8) field and was analyzed in detail on an Intel i5-2500K, 3.3 GHz, 8 Gbit RAM, computer system. The achieved throughput for sharing large files for an applied 5/3 threshold scheme was 7.4 Mbit/s. This value is relatively small compared to Gigabit networks and small extension fields.

In [7] secret sharing schemes were analyzed in terms of their performance. The focus of their work is the comparison of different schemes according to the threshold parameters n and k. There was no information about the used specific computer system. However, with Shamir's Secret Sharing scheme a throughput of about 9 Mbit/s could be achieved for generating 10 shares. The Computational Secret Sharing achieved a throughput of about 18 Mbit/s for the same amount of shares.

To increase the throughput, efforts were made to use the Graphics Processing Unit (GPU) for better performance. In [13] it was shown that the benefit of a GPU increases with higher thresholds. A threshold of 4 achieves a throughput of 48 Mbit/s for the calculation of one share. Another implementation of secret sharing based on cellular automata on a GPU [38] reveals a speed of 40-160 Mbit/s in a 5/5 threshold scheme.

3 Shamir's Secret Sharing

In this section the implementation of Shamir's Secret Sharing scheme in a FPGA is presented. Its resource utilization is discussed as its performance is evaluated against a software implementation as well.

3.1 Introduction

The concept of Shamir's Secret Sharing scheme was introduced in chapter 2.3. The aim of this chapter was to find an efficient FPGA implementation of Shamir's scheme in terms of throughput.

In the use-case of this work the secrets to be shared are files. In order to apply Shamir's algorithm, first the secret has to be divided into words of a certain bit-width depending on the selected binary extension field. The binary extension field is generic and can be freely selected from $\mathrm{GF}(2^n), n \in \{8, 16, 32, 64, 128\}$ to compare their efficiency.

The tasks are divided into the share generation, the share reconstruction and finally the creation of a whole system which manages the dataflows. Different implementation strategies and optimization methods are presented and evaluated as is the efficiency of the algorithm, applied in different Galois fields. The results are fully parametrizable cores for the parameters (n, k) of a threshold secret sharing scheme and the bit-widths of secret-words.

3.2 Share Generation

The generation of a share according to Shamir's scheme requires the definition of an unique Shamir polynomial for each secret (-word) and the evaluation at a specific point x with the condition $x \neq 0$. The information required to build the Shamir polynomial includes the secret itself, the x-point and various random data for the coefficients $c^n, n > 0$.

While the randomness of the coefficients is of crucial importance to the secrecy of the scheme, the generation of random numbers is not covered in this chapter. Moreover, they are considered as given from an interface from an outer party. However, it is theoretical possible to include the random number generation within the FPGA even if it contradicts the FPGA's purpose, as it should be a fully determined system. Various works have identified sources of randomness on a FPGA and showed the possibility to build a TRNG within it, e.g. [51], [72], [66], [32]. In chapter 6.2 the generation of random numbers on a FPGA will be discussed.

The evaluation of a polynomial with a degree of n in a straight forward approach (3.1), processing the coefficients in an ascending manner, consists of $\sum_{i=0}^{n}(i)$ multiplications and n additions. The usage of the Horner scheme, where the coefficients are processed in an opposite way from c_n to c_0, allows a significant reduction to n multiplications and n additions. It is illustrated in equation (3.2). Additionally, it allows efficient implementations in hardware.

$$y = c_0 + c_1 \times x + c_2 \times x^2 + ... + c_{n-1} \times x^(n-1) + c_n \times x^n \tag{3.1}$$

$$y = (((c_n \times x + c_{n-1}) \times x + ... + c_2) \times x + c_1) \times x + c_0 \tag{3.2}$$

If the result of an operation exceed the bit-widths of its field, a reduction is required. Therefore three different operation types can be identified for the computation of the equation (3.2). They consist of an addition, a multiplication and a reduction.

Addition

The addition in a binary extension field is implemented efficiently with little effort. Due to the fact that only two symbols are used for one digit in the polynomial, no carry bits are needed. The truth table for a bitwise addition is shown in table 3.1b and is the exact equivalent to a XOR-operation. The implementation on the FPGA could be realized with linear complexity in dependence of the bit-width of the operands.

+	0	1
0	0	1
1	1	0

(a)

×	0	1
0	0	0
1	0	1

(b)

Table 3.1: Truth table for bitwise addition and multiplication in GF(2^n).

Multiplication

FPGAs usually contain multiple DSPs to perform arithmetic operations. Because the multiplication in a binary extension field is different to an arithmetic operation, the DSPs cannot be used and specially designed GF(2^n)-multipliers are implemented with LUTs. In finite field arithmetic a multiplication consists of shift operations and additions. The difference to an arithmetic multiplier lies in the additions and the fact that there are no carry bits. The truth table of a bitwise multiplication in a binary extension field in shown in table 3.1a and is identical to an AND-operation.

The nature of the multiplication gives the possibility of implementing it sequentially by shifting, bit-wise multiplication and adding all sub-results together, or executing the whole functionality in parallel. The focus of this work is on the parallel implementation in order to gain a higher throughput and a more efficient use of LUTs within the FPGA. The resulting structure for a 3×2 polynomial multiplier is shown in figure 3.1, where the bitwise multiplication is performed with an AND-operation and the addition with a XOR-operation.

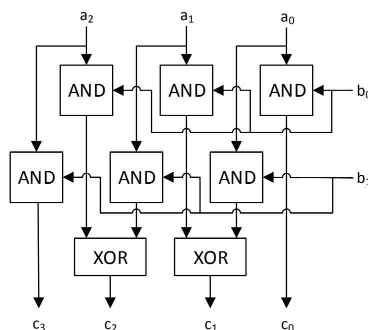


Figure 3.1: Parallel 2×3 bit polynomial multiplier architecture, build with logic gates.

The size of the multiplier increases with the square of the input size, assuming both inputs with identical widths. It is illustrated in figure 3.1. The growth in size strongly affects the processing of wider words, such as 128 bit. Furthermore, due to the longer paths in parallel architectures the minimal possible clock (clk) period increases. In table 3.2 the multiplier sizes after synthesis and implementation and the reachable minimum clock periods are summarized for the investigated bit-widths.

bit-widths of multiplicands	LUT synthesis	LUT implementation	minimal clk period [ns]
8	29	33	2.2
16	118	133	2.7
32	450	483	3.5
64	1729	1747	4.6
128	7175	7195	5.6

Table 3.2: Size of a GF(2^n)-multiplier in dependency of the bit-widths from the inputs, measured in 6-input LUT.

By limiting the possible x-values to a certain bit-width a decrease in size can be achieved. This method was already used before in [77]. It has no impact on the secrecy of the scheme, but it restricts the amount of possible generated shares since every share needs a different x-value to be unique. In table 3.3 the resulting size after synthesis of an asymmetric multiplier is shown.

As shown in table 3.3 the restriction of the x-values comes with a high reduction of LUTs. Limiting the possible x-values to results of the power of two, $x := \{2^n\}, n < bit - width$, an even higher decrease of LUTs could be gained. It would reduce the multiplication effort to the minimum, as it would result in processing only shift operations, which could be realized without any logic and only wiring. The main disadvantage is the relatively small set of possible x-values (8 in GF(2^8), 128 in GF(2^{128})) resulting in a small amount of different generated shares.

Another way of restricting x-values includes the usage of predefined static points. A multiplier with a static value fixed on one input results in a high decrease of required logic, but leads to the loss of flexibility at run time.

The final size of such a multiplier is strongly dependent on the actual fixed value. To gain an idea of its complexity, table 3.2 presents synthesis results with one value fixed at a specific point. In GF(2^8) all 255 possible values were synthesized, in GF(2^{16}) to GF(2^{128}) 200 values, equally divided over the corresponding field, were synthesized.

bit-widths	first multiplicand 8	16	32	64	128
2	6	11	19	33	65
4	16	32	64	128	208
6	21	41	81	159	319
8	27	51	99	224	386
12		78	170	317	577
16		114	200	392	775
20			284	559	1117
24			337	678	1327
28			394	777	1535
32			450	882	1742
48				1304	2602
64				1731	3520
96					5361
128					7239

(Rows: second multiplicand)

Table 3.3: Size of an asymmetric GF(2^n)-multiplier according to the bit-widths of its inputs respectively, measured in 6-input LUT.

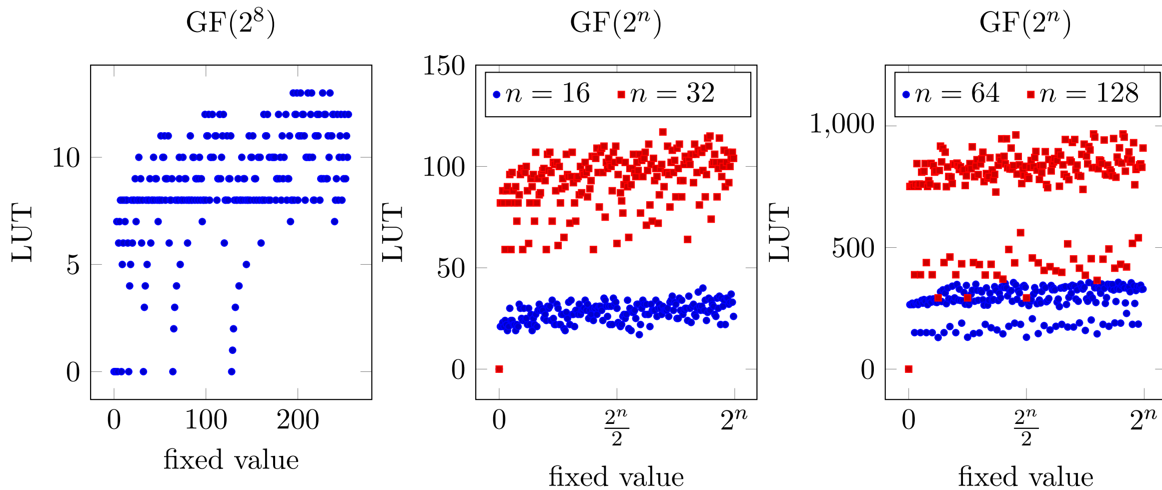


Figure 3.2: Size of a polynomial multiplier with one input fixed at a specific value (x-axis) for different Galois fields, measured in 6-input LUT.

For the rest of this thesis the x-value was restricted to 8 bit, resulting in 255 possible shares. The size was considered to be sufficient in a typical cloud based data storage setting and the decrease of up to 5.3% of originally needed LUTs in GF(2^{128}) brings high performance benefits as well.

Reduction

The reduction in a binary extension field accords to a modulo operation, where the value to be reduced is divided by the irreducible polynomial. The irreducible polynomial must be of degree n in a GF(2^n). Usually a set of irreducible polynomials are available for a binary extension field

and from a mathematical point of view all are equally suitable for defining the field. From the practical point of view, the weight of the polynomial and the number of nonzero coefficients have a direct influence on the complexity of the modulo operation and its hardware implementation. To implement one static irreducible divisor polynomial results in even more reduction potential of the circuit.

Such a modulo operation by a static divisor can be implemented efficiently as a linear feedback shift register (LFSR). The number of nonzero coefficients in the divisor polynomial equals the amount of XOR-connections in the feed-back loop. To that end, this approach needs n cycles to generate the results, where n is the number of input bits.

The polynomial division and modulo operation is a common problem, as it is used for many other applications, for example in the cyclic redundancy check (CRC). Consequently the polynomial division was part in a lot of research to investigate the realization in less clock cycles. In [81] a parallel implementation is described, which calculates the result within one clock cycle by using only static XOR-connections. The algorithm to find these connections was used for this work and is described below.

1. Consider the input data D of n bits as an $1 \times n$ vector and the irreducible polynomial as an $1 \times m$ vector.
2. Calculate n modulo results subsequently for the input data set $2^i, i = 1...n$, and store the bits of each result in a vector V_i. There will be i vectors V with the size $1 \times m$.
3. Create a $n \times m$ matrix M, where each column consists of the vector V.
4. Store the product of the matrix M and the input data D in a vector R, $R = M \otimes D$, which is the result. The symbol $\otimes$ denotes a matrix multiplication, where the multiplications are performed by AND-operations and additions by XOR-operations. As a result only those input bits remain for each output bit where the corresponding bit was set in the matrix M. Only those bits have to be XOR connected in hardware in order to produce the remainder R.

A Java program was developed in order to pre-calculate the required XOR-connections subsequently for all implemented fields. The resulting XOR-connections can be found in the appendix.

As mentioned before, the size of the chosen irreducible polynomial influences the complexity of the circuit. In [62] small irreducible polynomials were investigated for different binary extension fields. For each investigated and developed field in this work (GF(2^8), GF(2^{16}), GF(2^{32}), GF(2^{64}) and GF(2^{128})) the irreducible polynomial was chosen from [62] and a summary can be found in table 3.4.

bit-width	irreducible polynomial
8	$x^8 + x^4 + x^3 + x$
16	$x^{16} + x^5 + x^3 + x$
32	$x^{32} + x^7 + x^3 + x^2$
64	$x^{64} + x^4 + +x^3 + x$
128	$x^{128} + x^7 + x^2 + x$

Table 3.4: Used irreducible polynomials in this work.

The resulting size of the reduction circuit, measured in 6-input LUT, is shown in table 3.5 for each investigated Galois field. The input bit-width also influences the size, here input bit-width $2 \times (n-1)$ is assumed, which corresponds to a polynomial multiplication output for n input bits.

Galois fields Input bit-width	GF(2^8) 15	GF(2^{16}) 31	GF(2^{32}) 63	GF(2^{64}) 127	GF(2^{128}) 255
LUT needed	14	36	32	127	253

Table 3.5: Size of the reduction circuit in different Galois fields, measured in 6-input LUT after synthesis.

Share Generation

The character of equation (3.2) is best suited for a sequential computation in a feed-back loop, shown in figure 3.3b. One multiplier is used for performing all required multiplication steps subsequently. After each multiplication a reduction has to be performed, followed by an addition. The benefit of such an architecture are the arbitrary selectable threshold values at runtime, as only the amount of needed clock cycles changes. A drawback of such a scheme is the fact, that all operations have to be performed within the same cycle for an efficient capacity utilization of all units, because of the feed back loop. This leads to the disability of efficient pipelining.

On the other hand, an architecture fully capable of pipelining is shown in figure 3.3a, where architecture (b) is unrolled. In comparison to (b) the threshold value has to be chosen at synthesis time, which strongly effects the synthesis result in terms of space. Moreover, the data management becomes difficult. To alleviate data management issues, the design was build with a sequential approach.

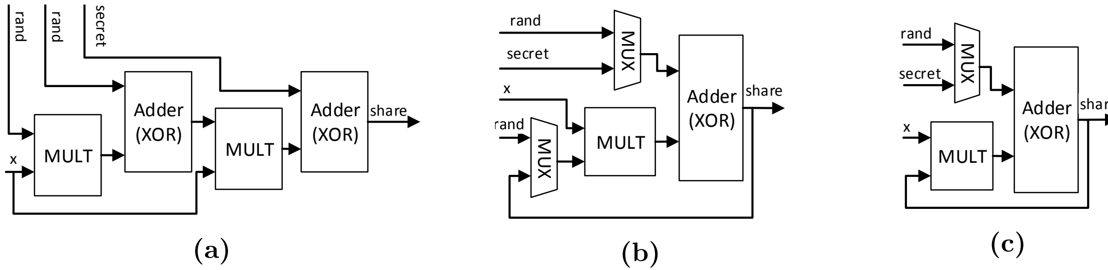


Figure 3.3: Multiple architectures for a share generation. (a) presents a pipelined architecture. while (b) and (c) are sequential architectures with a feed back loop.

As shown in figure 3.3b, two multiplexers are needed for the share generation. The value of the first summand has to be selected between the random values and the secret itself. On the other hand, the multiplier is fed with a random number in the first cycle of an evaluation round and the previous addition result in all other cycles. In this work a slight adjustment to this architecture was made, shown in figure 3.3c, which consists of only one multiplexer. Obviously this comes with the cost of the multiplier not being used the in first cycle of the evaluation. However, the data management becomes easier and one random value or secret is applied at a time, thus providing many advantages as well. It is of special benefit for looking up the next step of Computational Secret Sharing. Moreover, it becomes more LUT $\times$ number of cycles efficient for threshold values above a certain limit.

While architecture (b) needs $k-2$ cycles to calculate a share, architecture (c) needs $k-1$ cycles, where k is the threshold value in the secret sharing scheme. Higher values for k decrease the relative difference between these needed cycles and the saving of the second multiplexer is of greater influence. The relation is demonstrated in figure 3.4, exemplary for Galois fields GF(2^{16}) and GF(2^{64}), where architecture (b) becomes more LUT $\times$ cycle efficient after a certain threshold value k of 4 or 5.

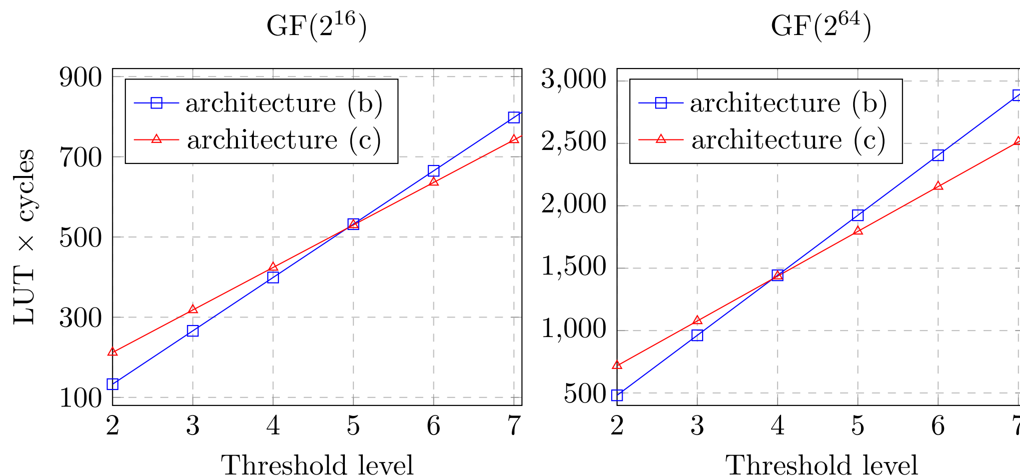


Figure 3.4: A comparison of the performance from architecture (a) and (b) for the share generation in dependence of the threshold value from the secret sharing scheme. The performance is measured in the cycle LUT product.

In a secret sharing setting for cloud applications, threshold values above or equal to four are expected. Combined with better management and adaptability to the Computational Secret Sharing scheme, architecture (c) was favoured and implemented.

The architecture of the final Share Generation Unit (SGU) is described in the following and presented in figure 3.5. In this work, the design is parametrizable for the used Galois field, the bit-width of the x-values and the number of parallel share generations. A finite state machine (FSM) controls the sharing process, while multiple Polynomial Evaluation Units (PEU) can be used to generate multiple shares at a time. Each PEU contains only elements that are needed to be unique for each share. It is beneficial to construct them parallel, as all need the same random and secret values at a time, which leads to only one required multiplexer for all PEUs. Moreover, the random values don't have to be saved for later rounds.

If a share construction is requested, the signal *enable* is set to high. The circuit starts to load the secret into a register and the first random word, while both read operations are confirmed via the corresponding *rand_rd* and *secret_rd* signal. Next, the FSM controls the multiplexer in order to distribute the random values for $k-2$ cycles and the secret at the last round. After the last round the corresponding share is at the output and the *share_valid* signal confirms the share. The register at the multiplier input is directly reset in preparation of the next round. The next round is initiated if the enable signal is high again.

The resulting size in terms of 6-input LUT, as well as the minimal possible clock period is demonstrated in figure 3.6, for the construction of 10 shares in parallel. Remarkable is the slight increase in resources and the decrease of the clock period for GF(2^{16}) compared to GF(2^{32}), mainly because of the efficient implementable irreducible polynomial in this field.

Figure 3.5: Architecture of the Share Generation Unit.

3.3 Secret Reconstruction

For the recovery of the original secret an interpolation is applied. In [77] various interpolation techniques were analysed in terms of their complexity and scalability. It was shown that the Lagrange interpolation is most suitable for this application. The Lagrange interpolation is performed in two steps. First basis-polynomials are computed as in (3.3) regarding the available points. By using the the shares as weights for the basis-polynomials, the y-value at the interested point is reconstructed by multiplying each shares with its corresponding basis polynomials and adding products together, shown in (3.4).

$$l(i) := \prod_{\substack{0 \leq m \leq k, \\ m \neq j}} \frac{x - x_m}{x_j - x_m} \tag{3.3}$$

$$L(x) := \sum_{j=0}^{k} y_i l_i(x) \tag{3.4}$$

The basis polynomials remain unchanged as long as points from the same x-value, regardless the y-value, are used. However, every time different points are included in the interpolation, all

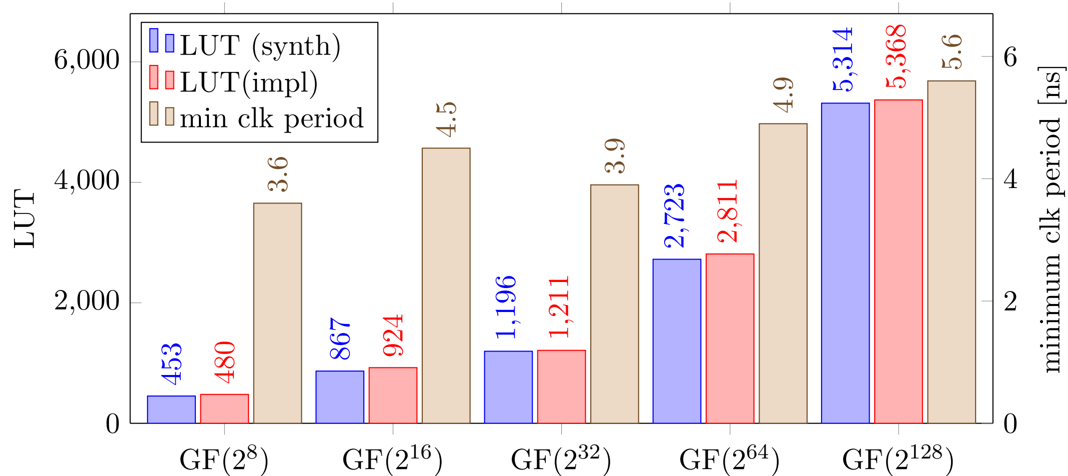


Figure 3.6: Size of the SGU in 6-input LUT for the synthesis and implementation and the minimal possible clk period. In the evaluated design 10 shares were constructed parallel.

of the basis polynomials have to be recalculated. At this point a practical assumption for a high performance increase is applied.

It is assumed that the applied x-values don't change with every processed word. This comes from the assumption, that all shares, stored on one specific server contains the same x-value and data in a network setup are usually transmitted in packets. The size of an Ethernet frame is typically about 1500 bytes, which would accord in GF(2^{32}) to about $1500/4 = 375$ words. When reconstructing a file, the source of the shares are not changed at a per word basis. They would at least change with every packet or more realistic they won't change during a hole reconstruction process. The distributed setup is due to security and reliability reasons and there is no need to change servers frequently, except if required by outer circumstances. For that reason, the computational intense calculations of the basis polynomial are outsourced to a processing system (PS) within the FPGA, and the available logic is used for computations continuously.

The implementation of the resulting operations into logic are addition, multiplication and reduction, similar to the previous section of the share generation. The operations addition and reduction remain unchanged. The multiplication optimization methods applied for the share generation are not applicable here, because a symmetric multiplier is necessary. However, there are multiple methods for further reduction of the size of a polynomial multiplier. This will be discussed in more detail in chapter 4. For the future design a multiplier with several, so called, Karatsuba levels is used. These designs show a significant increase of the minimal clock period for higher bit-widths. However, because the multiplier is not used in a feed-back-loop, it can be efficiently pipelined to overcome timing problem.

The weighting function of the shares with the basis polynomial consists of multiplications and additions. A use of k multipliers, illustrated in figure 3.7a, enables an efficient reconstruction of one share per cycle. However, it is possible to use only one multiplier and apply the coefficients subsequently, shown in figure 3.7b. An efficient pipelining of the multiplier is implemented since only an adder is in a feed-back-loop.

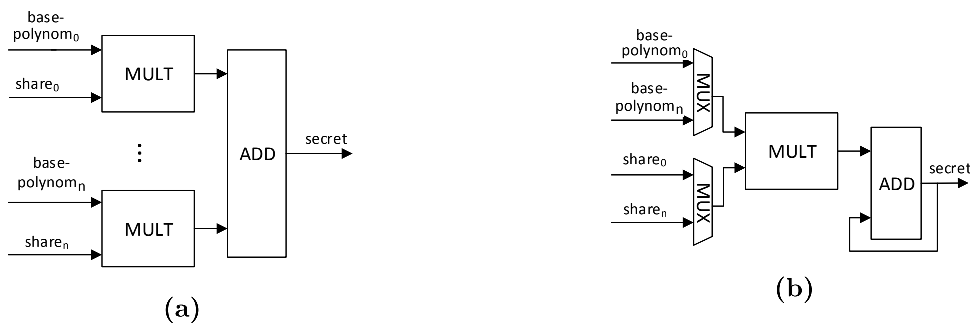


Figure 3.7: A sequential and parallel architecture for the secret reconstruction.

The architecture, as shown in figure 3.7b is selected in order to reach the same share generation and secret reconstruction speed as described below. It was designed generically to be able to work in all discussed Galois fields. A generic number of Karatsuba levels are applied in the multiplier and the opportunity to choose pipelining or not. The final architecture is shown in figure 3.8. The PS is the head of the reconstruction process and managing the computation of the basis polynomials. The basis polynomials are computed in the PS using a C-library from James S. Plank [26]. After computation the basis polynomials are loaded into the corresponding hardware registers via an AXI bus. The shares are supplied subsequently over the same bus. This assumption was taken to facilitate the circuits, but can be easily extended with a set of share registers and a multiplier.

If shares are available, the Secret Reconstruction Unit (SRU) is activated by setting the enable signal to high. The first base polynomial is now forwarded to the multiplier by the multiplexer. Additionally, the first share is aligned to the multiplier, and its read is conformed by setting the *share_read* signal to high. The register at the adder input is reset according to the number of pipelining stages if the first multiplication result of the new reconstruction process is available on the adders input. With the input *threshold*, the threshold level is applied to the circuit. In the following cycles all weighted base polynomials are added together successively. Finally, the *secret_ready* signal is set to high, signalizing the final reconstruction of the secret.

The resulting size in 6-input LUT and the achieved minimum clock period with the architecture is shown in figure 3.9 for a pipelined and non-pipelined version respectively. Due to the symmetric multiplier the size of the final Secret Reconstruction Unit increases quadratically. The influence of the pipelining stages results in a slightly bigger final design, because less logic optimization is possible. On the other hand, a significant increase of the possible clock frequency is possible.

3.4 Evaluation

In order to evaluate and compare the presented implementations, both a secret sharing software implementation was compared to this hardware implementation and a full working FPGA implementations is realized, which communicates with external systems and is capable of secret sharing and secret reconstruction.

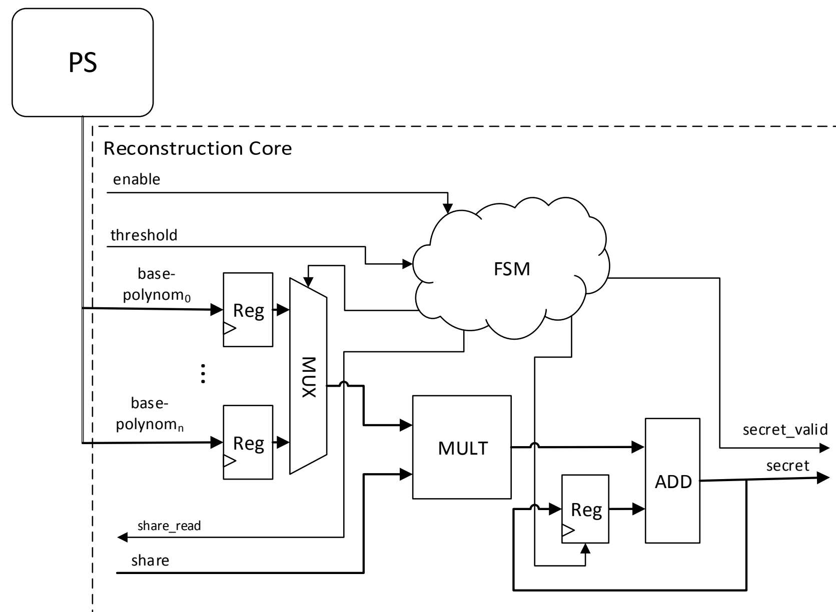


Figure 3.8: Architecture of the Secret Reconstruction Unit.

Hardware Software Comparison

For the evaluation against a computer program, a simple Shamir Secret Sharing algorithm was implemented, which can operate in in all investigated Galois fields. Within this program, all additional tasks, such as the random number generation, read and write procedures to the hard disk or network and other data management, were excluded. This was in order to find the theoretical maximal throughput.

Various software libraries were tested in order to find an efficient implementation for the calculation in the Galois fields. The selected library for the implementation is from James S. Plank [26]. It contains three different multiplication strategies, each applied for different Galois fields. Fields up to 9 bit are implemented in look-up-tables, fields from 10 to 22 bit are implemented with logarithmic and inverse logarithmic look-up-tables and an addition and fields bigger or euqal to 23 bit are implemented with shift and addition operations. The software was running using multithreading and 100% utilization of each processor core on a computer with the following specifications:

1. Processor: Intel i5-4590, Quadcore, 3.3 GHz
2. RAM: 16 GB, DDR3
3. Operating System: Windows 7, 64 bit
4. Hard Disk: 512GB, SSD, Samsung 850 Pro
5. Thermal Design Power (TDP): 84 W

The hardware was implemented on the Zedboard using a Zynq®-7000 All Programmable SoC XC7Z020-CLG484-1 FPGA. The data management was neglected for the FPGA, but due to

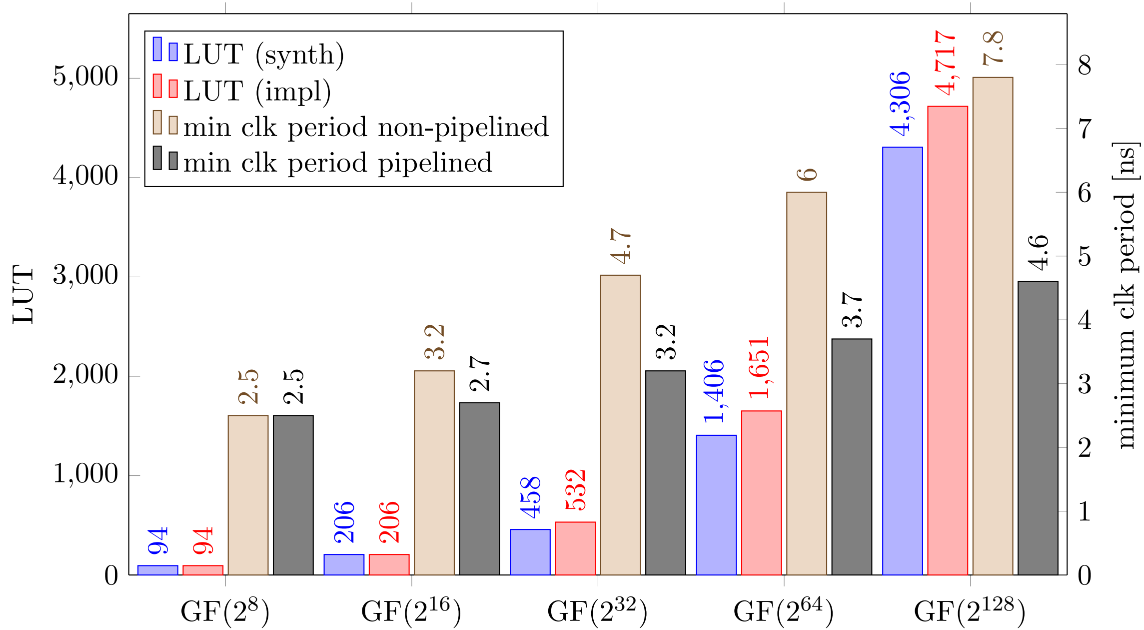


Figure 3.9: Size of the SRU in 6-input LUT, for synthesis and implementation as well as the minimal possible clk period. It is shown for both, a pipelined and non-pipelined architecture.

reservations of structures and routing, an utilization from only 50% was assumed, resulting in 26600 available LUTs. For a theoretical comparison, the architecture results from the previous sections were used and simply extrapolated for a usage of of 26600 LUTs.

The final results are shown in figure 3.10. A SGU with a parallel generation of 10 shares was applied in the FPGA design. The throughput is measures in Gbit/s according to the output of a functionality. This is generated share bits for the SGU and reconstructed secret bits for the SRU.

Figure 3.10 shows an almost constant speed of the share generation in the FPGA, with a single peak at 32 bit. This effect of constancy is caused by the static size of 8 bit for the x-value. In the computer implementation a strong drop of the performance is notable at 32 bit. At this point look up table based calculations are not applicable anymore and the relatively slow shift based multiplications are necessary. Overall the FPGA design turns out to have a performance gain of a factor of about 100.

Field	Share Generation		Secret Reconstruction	
	FPGA	PC	FPGA	PC
GF(2^8)	8.2mW	42W	9.9mW	50W
GF(2^{16})	10.7mW	26W	13.3mW	39W
GF(2^{32})	6.4mW	205W	22.1mW	275W
GF(2^{64})	9.0mW	237W	43.3mW	318W
GF(2^{128})	5.2mW	317W	89.1mW	429W

Table 3.6: Power consumptions for a throughput of 1 Gbit/s.

Based on the throughput shown in figure 3.10 an estimation for the power consumption per throughput was made. The Intel i5 processor has a power consumption of 84 W and the power

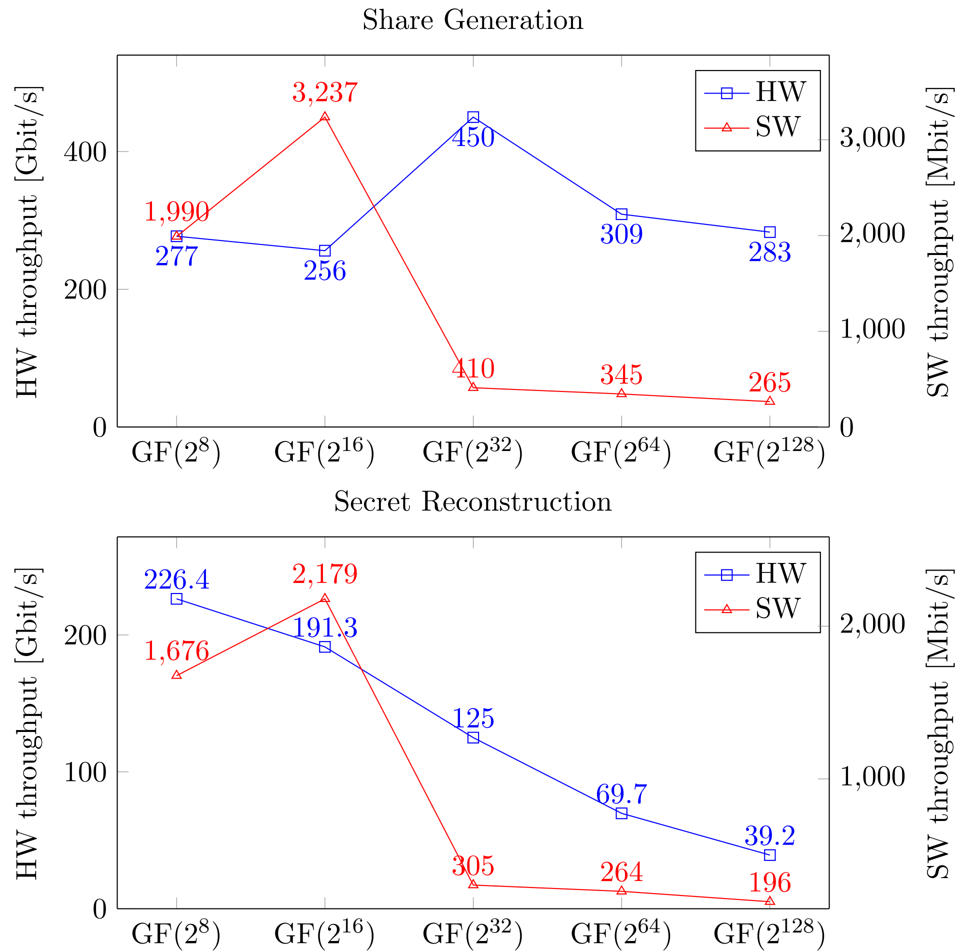


Figure 3.10: The maximal theoretical throughput of a software and a FPGA implementation, for the share generation and secret reconstruction respectively. A parallel construction of 10 shares in the FPGA is assumed. The throughput is measured in share bits per second for the share generation and secret bits per second for the secret reconstruction.

consumption of the FPGA was estimated using Vivado. In table 3.6 the resulting power consumptions for a throughput of 1 Gbit/s are compared.

Implementation of a complete System on the Zedboard

While the implementation for all investigated Galois fields were verified with simulation, for the practical proof of concept a fully working implementation on the Zedboard was made. For this purpose an AXI secret sharing intellectual property (IP_1) core was build, which communicates via an AXI bus. The PS takes care of the communication to the outer world and applies all necessary data to the secret sharing core. These data includes the calculated basis polynomial, the x-values, the threshold level, the secret and the shares. A simple control and status register is accessible for the PS to enable the SGU or SRU. All secret and share data were buffered within the SSS core using First In- First Out (FIFO) on all inputs and outputs. The status of the FIFOs were accessible by the control and status register. A simple TRNG was included in the design, its structure will be discussed in detail in section 6.2.

The implemented system works in GF(2^{32}) for simplicity reasons, as the AXI bus of the PS has a width of 32 bit. The final usage, including the logic of the FIFOs and communication is shown in table 3.7. 8 shares were produced parallel and a threshold value of 4 was selected. With a used clock frequency of 200 MHz, the implementation achieves a throughput of 12.8 Gbits/s for the share generation, measured in shared bits produced per second and 1.6 Gbits/s for the secret reconstruction, measured in secret bits recovered per second. The device utilization is about 5.8% of the available LUTs.

	SGU	SRU	TRNG	SGU-FIFOs	SRU-FIFOs	AXI Bus
LUT	934	390	384	375	293	597
BRAM				4.5	1	

Table 3.7: The FPGA implementation result for a SSS core, listed by functional units.

4 Advanced Multipliers

In this chapter the polynomial multiplier realization within a FPGA and various improvement strategies are analyzed.

First mathematical algorithms to reduce the computational complexity are presented and the Karatsuba algorithm is discussed and evaluated in more detail as well as its implementation in the FPGA. In order to further decrease the final LUT count FPGA specific strategies are investigated based on including other design elements of the FPGA.

4.1 Introduction

The implementation of Shamir's Secret Sharing reveals the multiplier to be the bottleneck of the design, with the highest consumption of resources. Therefore other implementation techniques were investigated in order to reduce the amount of necessary LUTs for a multiplier.

While the classical multiplication straight forward approach has a quadratic complexity of $O(n^2)$, with n as the number of input bits, various algorithms for the reduction of the complexity of a multiplication exist. They include the Karatsuba algorithm [40], the Toom multiplication [71] and its generalization, the Tom-Cook algorithm [16] as well as improvements of it [83], the Schönhage-Strassen algorithm [61], the Knuth algorithm [43] and algorithms using the Fast Fourier Transformation (FFT) [11] [17].

In comparison, the Karatsuba algorithm is best suited for a practical implementation due to its simplicity. For that reason a performance gain with a Karatsuba multiplier is discussed in many papers equally for soft- and hardware, e.g. in [29] and [28]. It is applicable to any kind of multiplication including finite field applications. In section 4.2 the influence of the Karatsuba algorithm in different binary extension fields is examined.

In section 4.3 alternative methods to decrease the utilization of LUTs are investigated, based on the application of addition design elements within a FPGA for performing a polynomial multiplication.

4.2 Applying Karatsuba's Algorithm

In 1963 Karatsuba published an algorithm to reduce the complexity of a multiplication, known as the Karatsuba algorithm. The idea follows the 'divide and conquer' scheme. A multiplication

with a complexity of $O(n^2)$ is broken down to smaller multiplications and additions, with a complexity of $O(n)$. The resulting overall complexity of this scheme is $O(2^{log_2(3)})$. Even with the theoretical improvement it is not guaranteed to reach the same complexity reduction within a FPGA resulting in a LUTs reduction. More complex data paths may result in longer delays. For this purpose the algorithm was investigated for a 6-input LUT. In the following procedure it is described how the Karatsuba algorithm works.

Consider two input multiplicands A and B with a bit-width of m. Both are divided into two parts with half of the bit-width A_h, A_l and B_h,B_l. A_l,B_l represent the lower bits and A_h,B_h the higher bits of A and B in equation (4.1). Mathematically, we denote such a split by a multiplication with R^m, where R represents the base of the used number system, e.g. 10 in the decimal system or 2 in the binary system, and m symbolizes the position of a single symbol within a number, e.g. $1R^1 = 10$.

We consider a and b written in lower cases as the digits of A and B respectively and the value x in the brackets (x) indicates the bit number.

$$\begin{aligned} A &= A_h R^{\frac{m}{2}} + A_l = a_m + ... + a_0 \\ B &= B_h R^{\frac{m}{2}} + B_l = b_m + ... + b_0 \\ A_h &= a(m) + ... + a(\frac{m}{2}), A_l = a(\frac{m}{2} - 1) + ... + a(0) \\ B_h &= b(m) + ... + b(\frac{m}{2}), B_l = b(\frac{m}{2} - 1) + ... + b(0) \end{aligned} \tag{4.1}$$

The product AB of A and B is expressed as the product of its sum.

$$\begin{aligned} AB &= (A_h R^m + A_l)(B_h R^m + B_l) \\ &= \underbrace{A_h B_h}_{P_a} R^{2m} + \underbrace{(A_h B_l + B_h A_l)}_{P_c} R^m + \underbrace{A_l B_l}_{P_b} \\ &= P_a R^{2m} + P_c R^m + P_b \end{aligned} \tag{4.2}$$

The facilitation is given by a different representation of P_c

$$\begin{aligned} P_c &= A_h B_l + B_h A_l \\ &= A_h B_h - A_h B_h + A_l B_l - A_l B_l + A_h B_l + B_h A_l \\ &= \underbrace{(A_h + A_l)(B_h + B_l)}_{P_d} - \underbrace{A_h B_h}_{P_a} - \underbrace{A_l B_l}_{P_b} \\ &= P_d - P_a - P_b \end{aligned} \tag{4.3}$$

with the new product term P_d. The final result is expressed as

$$AB = P_a R^{2m} + (P_d - P_a - P_b) R^m + P_b \tag{4.4}$$

and the three products P_a, P_b and P_d of two inputs with the width of $\frac{m}{2}$ and four additions or subtractions are performed. While the addition is of linear complexity and the multiplication is of quadratic, the resulting complexity is $O(m^{log_2(3)})$. With larger bit-widths m, the performance of a straight forward implementation increases.

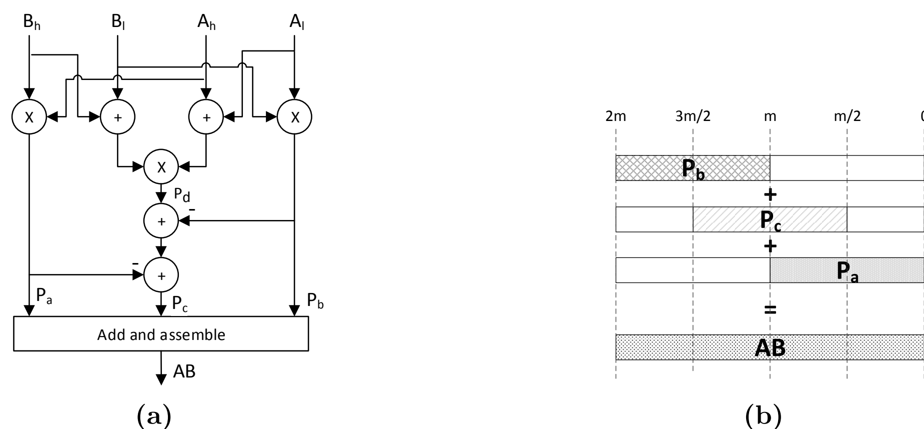


Figure 4.1: A symbolic representation of Karatsuba's algorithm is shown in (a). In (b) the addition and assembling regarding its bit position of the 'add and assemble'-block is illustrated.

Karatsuba's algorithm is well suited for a hardware implementation, e.g. shown in [75]. Figure 4.1a presents a graphical representation of the structure, which leads directly to a hardware architecture. In a binary extension field, the addition and subtraction is replaced by a XOR-operation and the multiplication is performed by a polynomial multiplier. The final 'add and assemble' block in figure 4.1a is in charge of assembling the signals P_a, P_b and P_c according to their bit position and adding them by a XOR-connection, illustrated in figure 4.1b.

In order to increase the performance gain, the facilitation efforts are applied recursively for all sub-multiplications in Karatsuba's algorithm. We refer to the amount of recursive applied Karatsuba multiplications as Karatsuba-levels, e.g. a Karatsuba-level of one stands for a single application of the formula. Due to the nature of logic synthesis and LUTs, the benefit of the Karatsuba algorithm starts to influence the design positively at a certain bit-width. At a bit-width of 8 it is of disadvantage to apply Karatsuba's formula. The utilization benefit of the formula starts at 16 bit and limits the performance gain of using multiple Karatsuba-levels. In figure 4.2 the size of multipliers according to its Karatsuba-levels is shown for the Galois fields $\mathrm{GF}(2^n)$, $n \in \{16, 32, 64, 128\}$, with $n = 8$ excluded as there are no benefits recognizable. The minimum clock period is shown subsequently for each level and an increase in time with ascending Karatsuba-levels is noticeable. This is due to a higher datapath dependency.

To overcome the increasing time effort, multiple pipelining stages were applied. They decrease the minimum clock period significantly, but come with the cost of a bigger design size, due to less optimization possibilities in the logic. The pipelining architecture presented in figure 4.2 consists of registers before and after each sub-multiplication. The double-register style was applied, because it showed the biggest size-to-time benefit. The total savings for an optimal Karatsuba-level rises with the bit-width of the inputs, summarized in table 4.1.

bit-width of the inputs:	8	16	32	64	128
optimal Karatsuba-level:	0	1	2	3	4
LUT savings:	0%	14%	16%	33%	47%

Table 4.1: LUT savings for an optimal Karatsuba-level compared to a straight forward multiplier architecture.

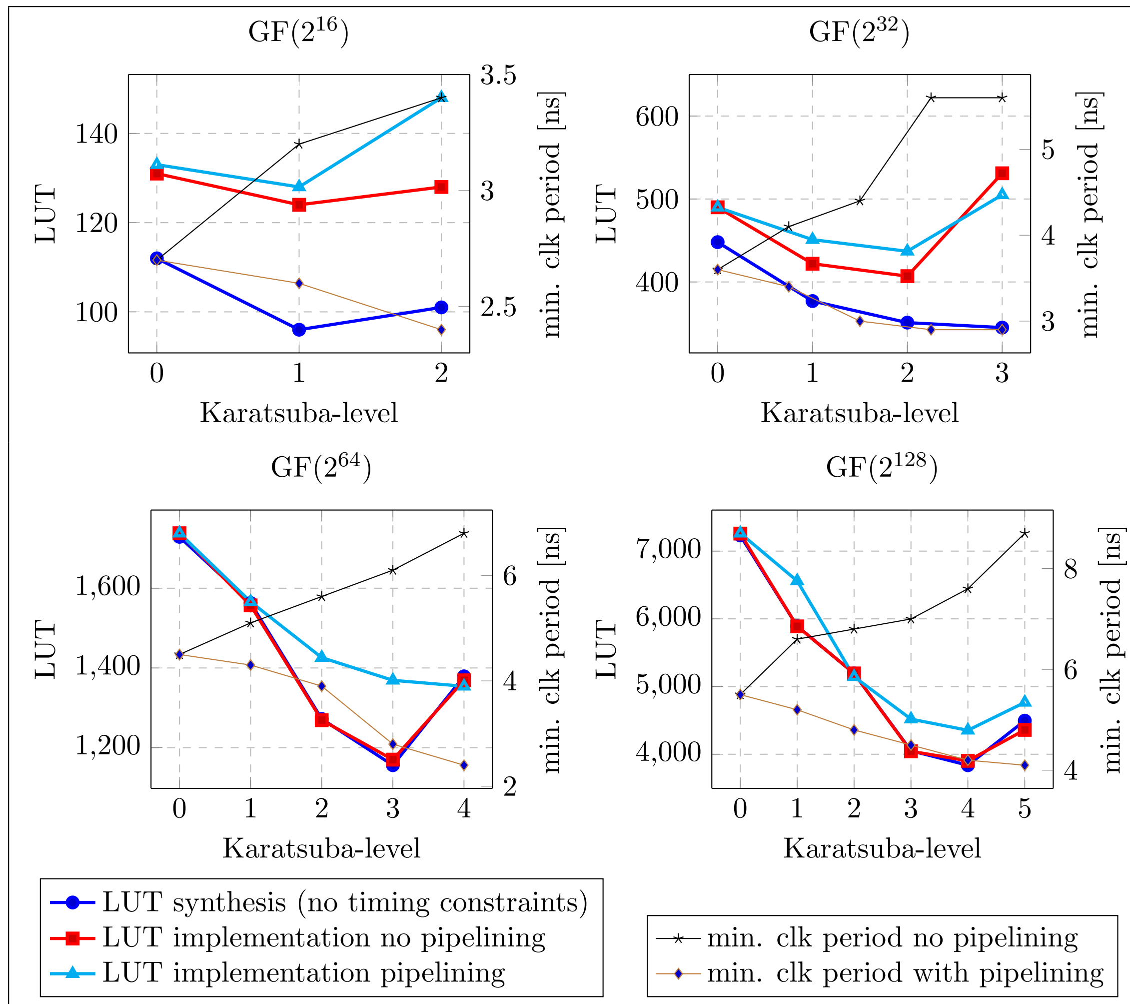


Figure 4.2: Multiple Karatsuba-levels applied for multipliers in different Galois fields. The resulting size in 6-input LUT of the synthesis as well as the implementation results, for both a pipelined and a non-pipelined version is illustrated.

4.3 A Better Overall Resource Utilization

After applying the Karatsuba algorithm and its strong improvements of LUT utilization, the multiplier remains the biggest block in the design and bottleneck. Therefore additional methods for decreasing the LUT count were investigated. Because the target architecture of this work is a FPGA and design elements other than LUTs are available, they were included in the multipliers design. The design elements on a FPGA are LUTs, D-FFs, DSPs and BRAM.

The following designs include these elements in order to reduce the LUT count and are evaluated in terms of their efficiency and usability.

D-FF

Alternatively the D-FF works similar to an AND-gate with additional buffering using the data and the enable input. The AND-gates in the polynomial multiplier, shown in figure 3.1, are

substituted by D-FFs resulting in the design of figure 4.3. This design is intended for a 3×2 bit multiplication of the multiplicands a and b with the output product c.

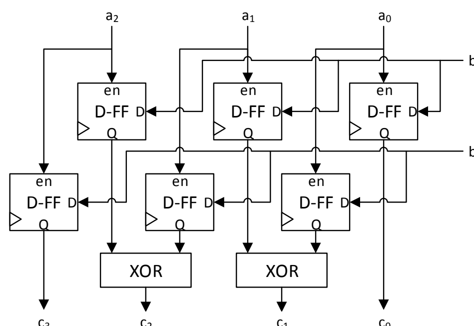


Figure 4.3: A 3×2 bit polynomial multiplier architecture, using D-FFs as AND gates.

In table 4.2 the resulting LUT and D-FF utilization of multipliers in different Galois fields, using D-FFs as AND-gates, are summarized. To give a relation for the amount of D-FFs, the percentage regarding a Zynq®-7000 FPGA with 106400 available D-FFs is recorded.

Even with a high amount of D-FFs within a FPGA, the required number is very extensive in this design, especially for multipliers of large bit-widths.

Field	LUT	D-FF	D-FF utilization	LUT savings
GF(2^8)	25	79	0.07%	0 (0%)
GF(2^{16})	74	287	0.27%	38 (33.9%)
GF(2^{32})	250	1087	1.02%	198 (44.2%)
GF(2^{64})	910	4223	3.97%	819 (47.4%)
GF(2^{128})	3457	16639	15.64%	3775 (52.2%)

Table 4.2: The size of a polynomial multiplier using D-FFs in the design.

BRAM

FPGAs usually contain BRAM within the logic area for fast data-storage. The Zynq®-7000 contains a 280 dual-ported 18 kbit BRAM block. A possible way of allocating the BRAM blocks to other operations is to store look-up-tables with data of the pre-calculated multiplication results. In a straight forward approach, the data is accessed using the input multiplicands as the address and the result is available on its data output. The result is either not-reduced or reduced according to the Galois field. However, in this case it is not possible to implement such a multiplier for a sub-multiplication for bigger Galois fields, e.g. as a sub-multiplier within the Karatsuba algorithm. The application of the sub-multiplication requires not-reduced results stored, which leads to a double memory size.

Another technical opportunity for look-up-table based multiplications with less memory storage applies the logarithm. By applying formula (4.5) the logarithm and inverse logarithm is pre-calculated and stored in a BRAM look-up-table.

$$ab = log^{-1}(log(ab)) = \underbrace{log^{-1}}_{\substack{BRAM\\LUT_1}}(\underbrace{log(a)}_{\substack{BRAM\\LUT_2}} + \underbrace{log(b)}_{\substack{BRAM\\LUT_3}}) \tag{4.5}$$

The outputs of the logarithm look-up-tables are added and used as input for the inverse logarithm look-up-table. The reduction of the memory amounts up to 3/4 of an approach storing the final multiplication results. However, such a multiplication is not applicable for a sub-multiplication in a bigger field. In table 4.3 the demanded memory is listed for the three architectures discussed above. The column "red." refers to a multiplier with stored reduced results, "no red." to stored unreduced look-up-table results and "log" to an architecture applying formula (4.5). Furthermore the required amount of 18 k-BRAMs in listed.

Of particular interest is that two multiplications are performed simultaneously with the same resource utilization a single multiplication would require. This is due to the dual ported structure of the BRAM.

While this method is reasonable for small Galois fields, it is not practical for bit-widths above 8 as the memory space would increase significantly.

bit-width	memory needed [bit]			18k BRAM blocks (utilization)		
of inputs	no red.	red.	log	no red.	red.	log
2	18	32	—	1 (0.4%)	1 (0.4%)	—
4	1792	1024	96	1 (0.4%)	1 (0.4%)	3 (1%)
6	45k	25k	3k	3 (1.1%)	1 (0.4%)	3 (1%)
8	983k	524k	73k	60 (21%)	32 (11%)	3 (1%)
10	20M	10M	157k	1152 (411%)	576 (202%)	3 (1%)
12	386M	201M	31M	—	—	9 (3%)
14	7G	3G	604M	—	—	39 (13%)
16	133G	67G	11G	—	—	180 (64%)

Table 4.3: The required memory size for look-up-table based multiplications and the effective amount of 18k bit BRAM for the realization.

DSP

A direct approach is not feasible using a DSP in a straight forward approach. However, a considerable amount of DSPs is available on a modern FPGA, e.g. 220 in Zynq®-7000, making it worth to include the DSPs into the final design.

The major difference in arithmetic and polynomial multiplication is the missing carry bit in a polynomial addition, which is performed subsequently in a multiplication process. The basic idea is to skip all bits possibly influenced by a carry bit. In other words, adding leads to bigger numbers in terms of digits representing these numbers. For example, by adding two binary numbers with the size of one bit, a maximum of one carry bit is produced for the case that both summands are one, $1_b + 1_b = 10_b$. If four one-bit numbers are added, the resulting number may need up to three bits, $1_b + 1_b + 1_b + 1_b = 100_b$. The last digit always accords to the result in a polynomial addition. Therefore not all of the input and result bits of a DSP are used and digits influenced by a previous carry bit are skipped. A resulting adding scheme for a 25×18 bit arithmetic multiplier, as it is available in a DSP of the Zynq®-7000 architecture, is shown in figure 4.4. The two polynomial input multiplicands are referred to as a and b. The multiplication is performed by shifting, bitwise multiplication and adding all sub-results together. The shift process is illustrated graphically, each row is bitwise multiplied by the input shown on the right, b, and then all rows are added to reach the final result c. Every third input bit of the DSP multiplier is connected

to a bit of a polynomial multiplicand. This leads to 6 inputs of multiplicand b and 9 inputs of multiplicand a, while all other inputs are hold at zero. The highest possible result of the addition in this scenario is 110_b, by adding six times one. Because only every third bit of the output is processed, the bits influenced by a carry do not influence the final polynomial multiplication result.

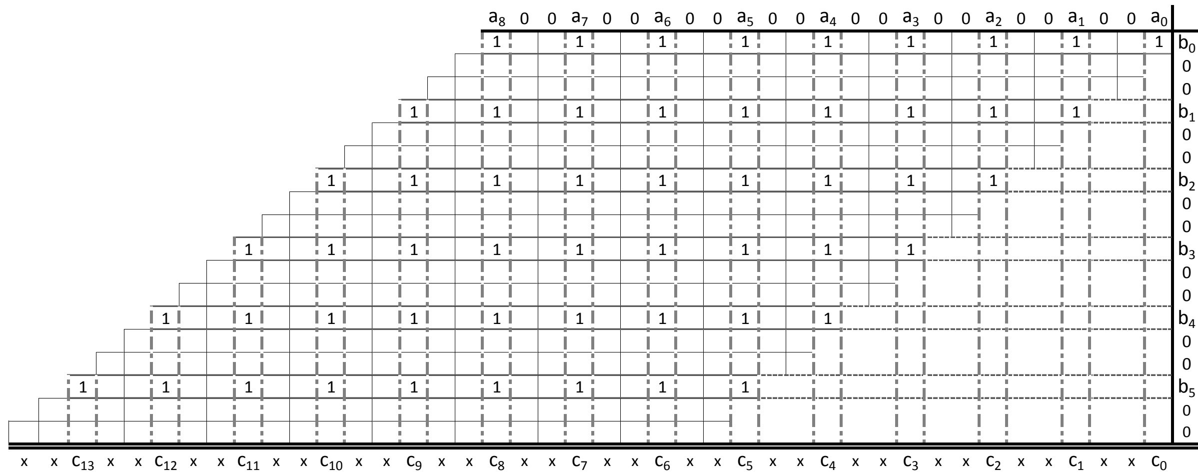


Figure 4.4: A schematic for performing a 6×9 bit polynomial multiplication in an 18×25 bit arithmetic multiplier (DSP), using every third bit, for the inputs (a_0 - a_8 and b_0 - b_5) and outputs (c_0 - c_{13}) respectively.

The final result is a 9×6 bit multiplier and for the same principle a multiplication of 13×3 bit is possible as well. This basic architecture was included in the designs for 8×8 and 16×16 bit multipliers, in order to integrate them as building blocks for multipliers of bigger sizes. In figure 4.5 the architecture for an 8×8 bit multiplier is shown. The multiplication of 8×6 bits is covered by a DSP multiplier, while the 2×8 bit multiplication is performed by LUTs. Both sub-results are added using LUTs. This leads to a total utilization of ten LUTs and one DSP, with 15 LUTs saved compared to a only LUT design.

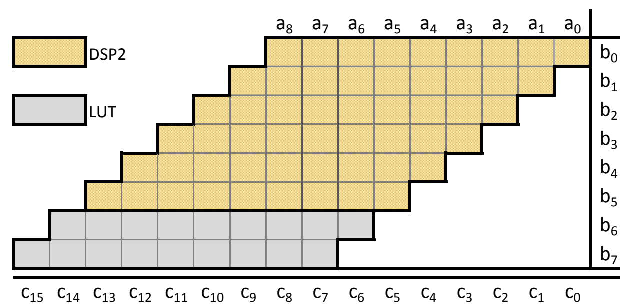


Figure 4.5: figure
The design for an 8×8 bit multiplier, with DSP and LUT

The 8×8 bit multiplier could be implemented as a basic building block in order to build bigger multipliers, such as a 16×16 multiplier by using them as sub-multipliers in Karatsuba's algorithm. However, more designs to build a static 16×16 bit polynomial multiplier were investigated, by including two to four DSPs into the design. In a straight forward multiplication design, some parts of sub-multiplications are performed by DSPs, while the rest is covered by LUTs. In table

4.4 the synthesis results are listed for multiple designs using different amounts of DSP, partly applying Karatsuba's algorithm.

Architecture:	(a)	(b)	–	(c)	–	(d)
	0 DSP + Karatsuba	2 DSP	2 DSP + Karatsuba	3 DSP	3 DSP + Karatsuba	4 DSP
LUT:	112	68	68	45	52	27
Saving/DSP:	–	22	22	22.3	20	21.25

Table 4.4: The utilization for a 16×16 bit multiplier including a different amount DSPs in the design and partly applying Karatsuba's algorithm.

Multiplier architecture (d) with four DSPs is demonstrated in figure 4.6 and distinguishes between whether the sub-multiplication is covered by DSPs or LUTs. The final sub-results are added using LUTs.

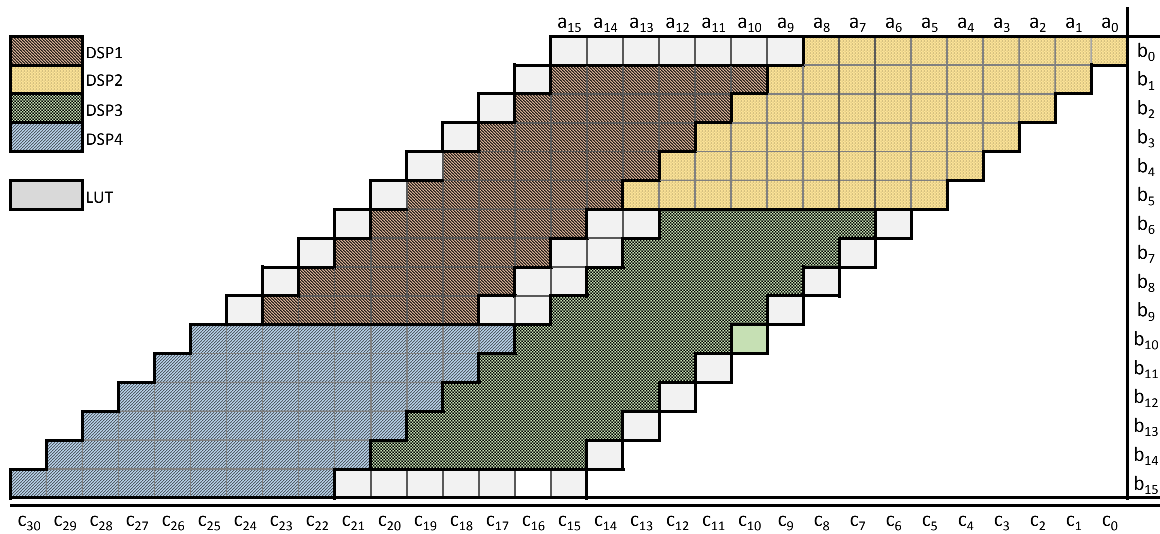


Figure 4.6: Architecture (d) for a 16 bit polynomial multiplier, realized with DSPs.

By applying both, the Karatsuba algorithm for breaking down a multiplication to 16×16 bit and using one of the architectures summarized in table 4.5, more complex multiplications are realized with DSPs in the design. In table 4.4 synthesis results for 32, 64 and 128 bit multipliers with DSPs are presented and compared for different basic 16×16 multiplication designs. The final design is fully generic, where one can choose which strategy to use.

On average 19 LUTs are saved per DSP. The target FPGA possesses 220 DSPs, which allows the saving of up to 3960 LUTs. With 53200 LUTs in the target FPGA, their utilization is reduced significantly by 8%.

4.4 Conclusion

Multiple methods to reduce the final LUT count of a multiplier were evaluated. The D-FF and BRAM based optimizations were not used, due to their marginal LUT savings by the high cost

16 bit mult. - architecture	(a)	(b)		(c)		(d)		min. clk period
	LUT	LUT	DSP	LUT	DSP	LUT	DSP	
32 bit	396	291	6	226	9	165	12	$\approx$ 3.3ns
64 bit	1236	996	18	808	27	622	36	$\approx$ 3.5ns
128 bit	4111	3233	54	2646	81	2121	108	$\approx$ 3.6ns

Table 4.5: The utilization of LUTs and DSPs using the Karatsuba algorithm until 16×16 bit multipliers are reached and an architecture from table 4.4 including multiple DSPs in the design.

of other design elements and the fact that they are highly needed for other tasks in the final implementation of a full system.

The DSP based optimization was included in the final multiplier design as a generic option, because its significant decrease of up to 8% of LUTs in the Zynq®-7000 architecture. Moreover, they have no other purpose. A disadvantage may be the resulting higher power consumption, as the DSPs are used in a very inefficient way. The Karatsuba algorithm is beneficial in wide bit-width multipliers and was included in the final design to apply this facilitation until a 16×16 bit sub-multiplication is reached.

5 Computational Secret Sharing

This chapter covers the progression of the implementation from Shamir's Secret Sharing to the Computational Secret Sharing scheme. It follows a theoretical speed comparison of a software implementation, and a full CSS core is developed, capable of share generation and secret reconstruction.

5.1 Introduction

The concept of Computational Secret Sharing was explained in chapter 2.4. The idea is to create a concept with minimal sizes of shares in order to achieve a better storage efficiency. This scheme was introduced in [46]. The scheme requires an encryption function, a decryption function, an informational theoretic secure secret sharing scheme (e.g. Shamir's scheme), and any secret sharing scheme with optimal storage efficiency.

The first step of Computational Secret Sharing is the encryption of the original secret. The encryption is performed with a random key, which is then shared using Shamir's Secret Sharing scheme. The encrypted secret itself is shared by applying the basic concept of Shamir's polynomial, but placing secrets instead of random numbers in all of the coefficients. The concept will be simply referred to as Extended (Shamir) Secret Sharing (ESS) scheme. The difference to Shamir's scheme is the absence of random numbers and thereby not putting redundancy into the shares itself. The result is an optimal storage efficiency for the ESS. The redundancy of the ESS shares is given only by the generation of more shares than necessary for the reconstruction. The key-share and secret-share are assembled into one message and sent to the shareholders. The key changes for every message and the message length is free to choose.

Nonetheless, the additional key-share needs to be stored as well and the size of all shares necessary for the reconstruction (which are k shares with k as the threshold value) asymptotically approaches the size of the secret in dependency of the message length.

The encryption and decryption function is performed according to the Advanced Encryption Standard from the National Institute of Standards and Technology (NIST) [2]. The standard is approved by the National Security Agency (NSA) for top secret information [54]. The algorithm is used worldwide, is publicly accessible and free to use.

The process of constructing shares according to the Computational Secret Sharing scheme with the AES algorithm for the encryption function is presented in figure 5.1. A TRNG generates

a random-key, which is utilized by the AES unit and shared with the Shamir's Secret Sharing unit as the key-share. The AES unit encrypts the secret with the random-key. Afterwards the produced ciphertext is shared with ESS unit and distributed as the secret-share.

The resulting SGU and SRU have the ability to operate in both modes, enabling the application of the same unit for key-shares and secret-shares, in order to reduce the logic.

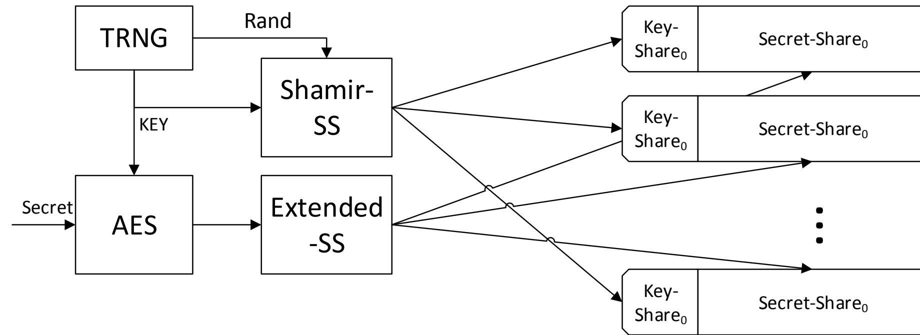


Figure 5.1: The theoretic concept of the Computational Secret Sharing generating the shares.

5.2 Advanced Encryption Standard

The AES is a block cipher, which is open and standardized by the National Institute of Standards and Technology in [2]. It is a subset of an encryption scheme developed by Joan Daemen and Vincent Rijmen in the year 2000 and published in [18] and [19]. Because it is a symmetric encryption scheme, the same key for encryption and decryption is used. The algorithm works with a block length of 128 bit, where the key size is 128, 192 or 265 bit. In dependency to the key length it is referred to as AES-128, AES-192, AES-256, but the block size always remains 128 bit.

How AES works

The AES algorithm operates with substitutions and permutations. Every 128 bit (16 bytes) block is expressed as a 4×4 matrix and all elements are of one byte size. Substitutions and permutations are performed on the elements in multiple rounds, where the key-size determines the number of rounds, 10 for a 128 bit key, 12 for a 192 bit key and 14 for a 256 bit key. In the following a basic description of an encoding in n rounds is explained:

1. Preparation: First, the key is expanded to obtain an unique key for the following rounds.
2. Round 0: (*AddRoundKey*) Every byte in the block is combined with a byte from the round-key using a XOR-operation.
3. Round 1 to $n - 1$: Multiple operations are performed:
   - (*SubBytes*) Every byte is substituted, using a non-linear substitution, which is stored in a look-up-table.

- (*ShiftRows*) The bytes of every row are shifted to the left, where the row number specifies how many times each byte is shifted.
- (*MixColumns*) Each column of the matrix is multiplied by a fixed polynomial, which can be stored efficiently in a look-up-table.
- (*AddRoundKey*) Finally, every byte is combined with its corresponding round-key as in round 0.

4. Round n: Operations similar to the rounds 1 to $n-1$, but the *MixColumns* step is not performed.

The operations mainly consist of look-up-table based substitutions, shift operations and simple XOR-operations. They enable efficient hardware and software implementations, investigated in [22] and [47], especially in comparison to other high-secure block ciphers [60]. The algorithm finds application in many practical implementations, e.g. the Wi-Fi Protected Access 2 (WPA2) encryption for wireless-LAN communication [41].

Block cipher modes

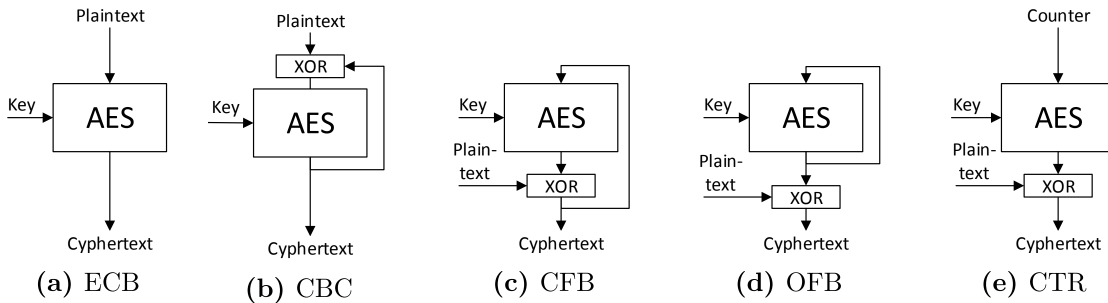
A block cipher, such as the AES, may reveal unwanted effects by directly feeding the plaintext to the input and process the output as ciphertext, even with no weakness of the block cipher itself. This comes from the fact, that each block is processed independently. Imagine two blocks with the same bit-pattern, e.g. all bits hold at zero. The corresponding output blocks will reveal an identical bit-pattern. An attacker may see if 128 bit blocks are identical, which could reveal structures of the original data.

To overcome this weakness, a change of the output pattern in dependency of the block-number or the previous processed block can be forced in order to guarantee a change in the ciphertext blocks, even with the plaintext blocks remaining the same.

The AES can be used in various modes. The NIST published five confidentially operation modes of a block cipher in [25], which are specified as well in a standard from the International Organisation for Standardisation (ISO) in [5]. These modes are summarized in the following and are illustrated in figure 5.2, where 5.2a is the basic AES mode which may reveal overlaying structures and the modes 5.2b to 5.2e overcome this problem by various strategies.

- Electronic Code Block (ECB): The plaintext is set as the input of the AES, while the output is directly used as ciphertext and illustrated in figure 5.2a.
- Cipher Block Chaining (CBC): The plaintext is combined by a XOR with the AES output of the previous round and applied as input for the AES in the current round. With a change of the output a input change is forced, which leads again to a changed output. The AES output is used as ciphertext and illustrated in figure 5.2b.
- Cipher Feedback (CFB): The ciphertext of the previous block is the input for the AES in the next round. The plaintext is combined with the AES output by a XOR-operation. This XOR-result is the ciphertext. Illustrated in figure 5.2c.
- Output Feedback (OFB): In comparison to CFB, not the ciphertext but the output of the AES is the input of the AES for the next round. Illustrated in figure 5.2d.

- Counter (CTR): The value of a counter is the input of the AES. In order to get the ciphertext, the AES output is XOR-combined with the plaintext. Illustrated in figure 5.2e.



(a) ECB **(b)** CBC **(c)** CFB **(d)** OFB **(e)** CTR

Figure 5.2: Different block cipher operation modes.

AES Implementation

Because various optimized FPGA implementations for AES exist, an intellectual-property-block of an AES core was implemented in this design. The criteria for such an core is to compete with the throughput of the rest of this design. A certified AES core from OpenCores [55] was selected, which is able to load one new 128 bit input block in each cycle. After a latency of 22 cycles, the encrypted block is available on its output, which leads to a throughput of 128 bit per clock cycle.

Furthermore, the core is optimized in speed and utilization. The non-linear substitution (*SubBytes*) and polynomial multiplication (*MixColumns*) is realized with look-up-tables in the BRAM. However, almost all of the 140 available BRAM elements on the Zynq®-7000 FPGA were utilized by the AES implementation. Coupled with little adjustments the BRAM count was reduced in order to save them for the later data buffering of the overall system. In table 5.1 the synthesis and implementation results of the AES core are summarized, including adjustments to reduce the BRAM count by performing *SubBytes*, *MixColumns* or both with LUTs.

AES-functions in BRAM	BRAM	LUT synthesis	LUT implementation	FF	min. clk period
No BRAM	0	11376	10919	5843	2.9ns
SubBytes in BRAM	36	8960	8960	5411	4.3ns
MixColumns in BRAM	50	7808	7593	4963	5ns
Both in BRAM	86	3200	2978	3811	5ns

Table 5.1: The synthesis and implementation results of the AES core, realizing different functionalities in BRAM based look-up-tables.

The AES core is operated in counter mode, which gives various advantages over the other operating modes. In comparison to ECB mode, it does not reveal overlaying structures of the whole data set. Compared to all other modes, there is no feedback loop of the previous output, which is not available before 22 cycles. Next, it is possible to start encrypting at an arbitrary block in a data set, because only the counter register needs to be preloaded with the according value. Moreover, encryption and decryption is performed with a completely identical unit. The AES output is the same for encryption and decryption. The ciphertext is generated by combining the AES output with the plaintext by a XOR-combination. On the other hand, combining the AES

output with the ciphertext by a XOR-connection reverses the first XOR-operation and reveals the plaintext.

The AES unit has a considerable high utilization of FPGA resources. For that reason, only one AES unit was build, which is shared between the share generation and secret reconstruction, as the full design should be capable of both functionalities. The resulting design is shown in figure 5.3. There are two data channels, for share generation and secret reconstruction respectively, which are multiplexed to the AES input. Two counter registers store the current counter value for each channel independently, while the correct counter value is selected by the control circuit. Via an external counter reset signal, the counters are reset. Two key-registers enable key-storage for both channels, while new keys are read whenever the counter is reset and the key corresponding to the processed channel is applied at the key-input of the AES core.

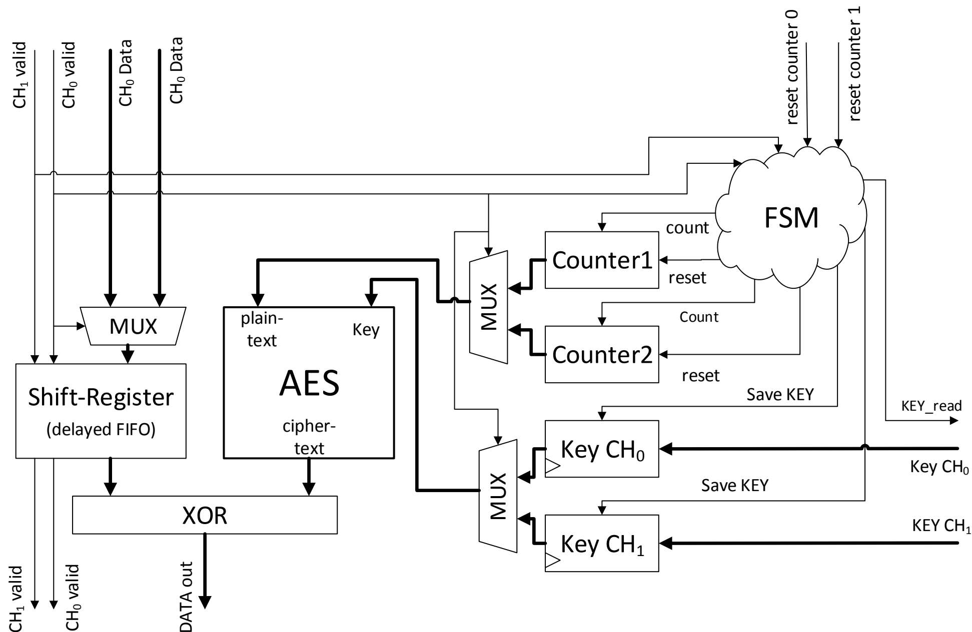


Figure 5.3: The architecture of the AES unit.

The secret is combined with the AES output 22 cycles after the restart of the counter due to the AES core latency. For simplification purposes the channel input data is stored in the AES unit for these 22 cycles. It allows the synchronous start of the timer when a new input is fed into the AES core. The 22 cycle delay and storage are selected generically for a D-FF based shift-register or a BRAM based FIFO-buffer.

The synthesis and implementation results of the AES unit are summarized in table 5.2. While the counter-width is generic, it was set to 16 bit. It allows up to $2^{16} \times 16byte = 1GByte$ for a dataset before the counter has to be reset. The shift register was implemented using D-FFs. A change to a BRAM based FIFO could save about $(128 + 2) \times 22 = 2860$ D-FFs at a cost of 2 additional BRAMs.

AES-Functions in BRAM	BRAM	LUT synthesis	LUT implementation	FF	min. clk period
No BRAM	0	10940	10912	9295	3.3ns
SubBytes in BRAM	36	9212	9182	8863	4.3ns
MixColumns in BRAM	49	7484	7537	8382	4.8ns
Both in BRAM	85	876	2913	7279	4.9ns

Table 5.2: The synthesis and implementation results of the AES unit (figure 5.3), realizing different AES-functionalities in BRAM look-up-tables. The counter width is 16 bit and the shift register was implemented using D-FFs.

5.3 Share Generation

The developed SGU for Shamir's scheme was slightly adapted. The only difference to Shamir's scheme is that the secret data is used for all coefficients. In the architecture presented in 3.5, a multiplexer selects whether the secret data or random values are directed to the Polynomial Evaluation Units. An additional signal *mode_select* is selecting between the SSS-mode and ESS-mode. In the SSS-mode the secret is positioned in the lowest coefficient c_0, while the multiplexer selects random words for the higher coefficients c_1 to c_n. In the ESS-mode the multiplexer always directs secret words to the Polynomial Evaluation Units.

5.4 Secret Reconstruction

The secret reconstruction in the Computational Secret Sharing scheme differs from Shamir's scheme by the amount of data extracted from the shares. While in Shamir's scheme only the coefficient c_0 of the Shamir polynomial is of interest, all of the coefficients are reconstructed in the Computational Secret Sharing scheme.

In chapter 3.3 the Lagrange interpolation was used to interpolate the y-value at a specific point ($x = 0$). However, in this scheme all coefficients of the Lagrange interpolation polynomial are determined, leading to a linear system of equations to be solved.

The evaluation process of the Shamir polynomial is a linear equation system. Written in matrix notation it leads to formula (5.1), with a matrix SEC containing a set of secrets, a matrix SH containing a set of shares and a matrix X with x-values and their powers. Such a set of secrets or shares are always computed together.

$$\underbrace{\begin{pmatrix} share_0 \\ share_1 \\ share_2 \\ share_3 \end{pmatrix}}_{SH} = \underbrace{\begin{pmatrix} x_0 & x_0 & x_0^2 & x_0^3 \\ x_1 & x_1 & x_1^2 & x_1^3 \\ x_2 & x_2 & x_2^2 & x_2^3 \\ x_3 & x_3 & x_3^2 & x_3^3 \end{pmatrix}}_{X} \underbrace{\begin{pmatrix} secret_0 \\ secret_1 \\ secret_2 \\ secret_3 \end{pmatrix}}_{SEC} \tag{5.1}$$

By solving the equation after SEC, the matrix X has to be inverted, which leads to equation (5.2) for the reconstruction of a set of secrets.

$$SEC = X^{-1} SH \tag{5.2}$$

The inversion of the matrix X depends on the x-values of the used shares. Similar to calculating the basis polynomial in Shamir's scheme, the operation is performed on the PS.

The calculation of $X^{-1}SH$ consists of additions and multiplications, similar to the weight-function of the Lagrange interpolation. While in Shamir's scheme k multiplication and $k-1$ additions are performed, this scheme requires k^2 multiplications and $(k-1)^2$ additions. Equation 5.2 is evaluated independently for every set of shares and therefore performed in the PL.

X-Matrix inversion

The inversion of the matrix is performed on the PS and was programmed in C. First, a matrix X has to generated, by computing the powers of the x-values. Then a simple and efficient algorithm for the inversion, published in [30], was implemented.

After processing the matrix, the coefficients are loaded subsequently into the PL, via an AXI bus.

Determining the secret matrix

The determination of the secret matrix is similar to the secret reconstruction in Schamir's scheme. One multiplier with a multiplexer on each input and an adder reconstruct the set of secrets subsequently. However, each set of secrets requires k cycles and leads to a slower reconstruction rate than for the share generation. A parallel approach was selected, mainly for achieving an identical throughput of the share generation and secret reconstruction, with the cost of loosing threshold flexibility at runtime. The threshold value is still generic, but set at synthesis time. The design is presented in figure 5.4.

Every row of the matrix X is calculated in parallel, where each multiplication takes place in a $share_n - subreconstruction$ block. The multiplication results are added and reduced in order to obtain one secret of the set. The whole process is pipelined efficiently since there is no feedback loop. The rows of the matrix are processed sequentially by applying different values of the matrix X, but the same share for each multiplier.

The values of the inverted matrix X^{-1} are loaded into the unit before activating the SRU. The interface for loading the values is generic for either a parallel port or a bus, as shown in figure 5.4. In the parallel design, the multiplexer selects all the matrix elements stored in a register. The bus design stores each column of the matrix X^{-1} in an individual memory. The memory is selected externally with the signal *sel_col* and all column elements are loaded serially via the bus *matrix_elements*.

The SRU is enabled via a logic high on the *enable* signal and starts to load a set of shares into the share buffers. It is conformed with a logic high on the signal *share_rd*. Immediately after the conformation the SRU starts the multiplication process, one row after another. The signal *row_sel* functions as the address for the column memories and selects the correct matrix elements for the multiplication input. After a certain amount of cycles according to the pipelining stages, the *secret_valid* pin is put to logic high signalising a valid secret on the output *secret*. The set of secrets appears immediately after each other on the output. One secret is reconstructed per clock cycle, if the shares are provided successively.

The reconstruction with Shamir's scheme for the key-secret is selected with the input *mode*, which switches between the SSS mode and the ESS mode. The SSS mode only reconstructs the

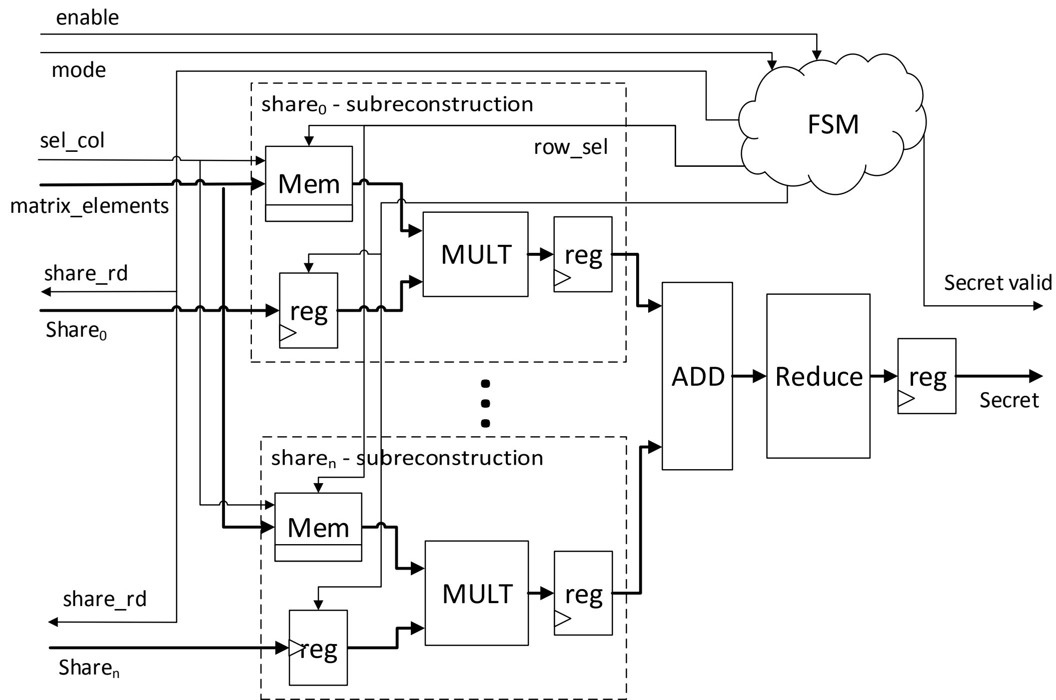


Figure 5.4: Secret Reconstruction Unit of the Computational Secret Sharing.

first row, equal to the coefficient c_0 in the Shamir polynomial. All other secrets in the set are random values and not necessary.

The size of this unit is shown in 5.5. A threshold value of 4 was assumed and the design is fully pipelined. Because of the high amount of LUTs needed for the four multipliers, the design was also evaluated with multipliers including DSPs as presented in chapter 4.

5.5 A Full Computational Secret Sharing Core

The functionalities of AES, SRU and SGU are put together in order to build a full CSS system. A packet oriented approach was chosen to allow a structured management of keys and data sets for the same key. The data are handled as packets and stored in packet buffers. An additional header information is used for the information if the share generation needs a new key, or a share packet contains a key-share. A simplified structure is presented in 5.6, where black lines represent data buses and grey lines buses for the header information.

In between the SRU and the AES, and the SGU and the AES a buffer is interconnected. The buffers are necessary as the SRU and SGU may operate at different speeds, depending on the Galois field. In a Galois field GF(2^{128}) the SRU and SGU are able to operate at the same speed as the AES. In GF(2^{64}) the AES multiplexes in between the SGU and the SRU with full time utilization of all units. The buffers also take care of the bit-width conversion from 128 bit of the AES unit to the bit-width used for the SGU, and vice versa. Additionally, the buffers are applied on all data inputs and outputs, capable of bit-width conversion as well. The AES-Arbiter takes care of a fair use of the AES in both directions. The channel is switched after each packet, if packets are available on the AES input buffer and the AES output buffers not full.

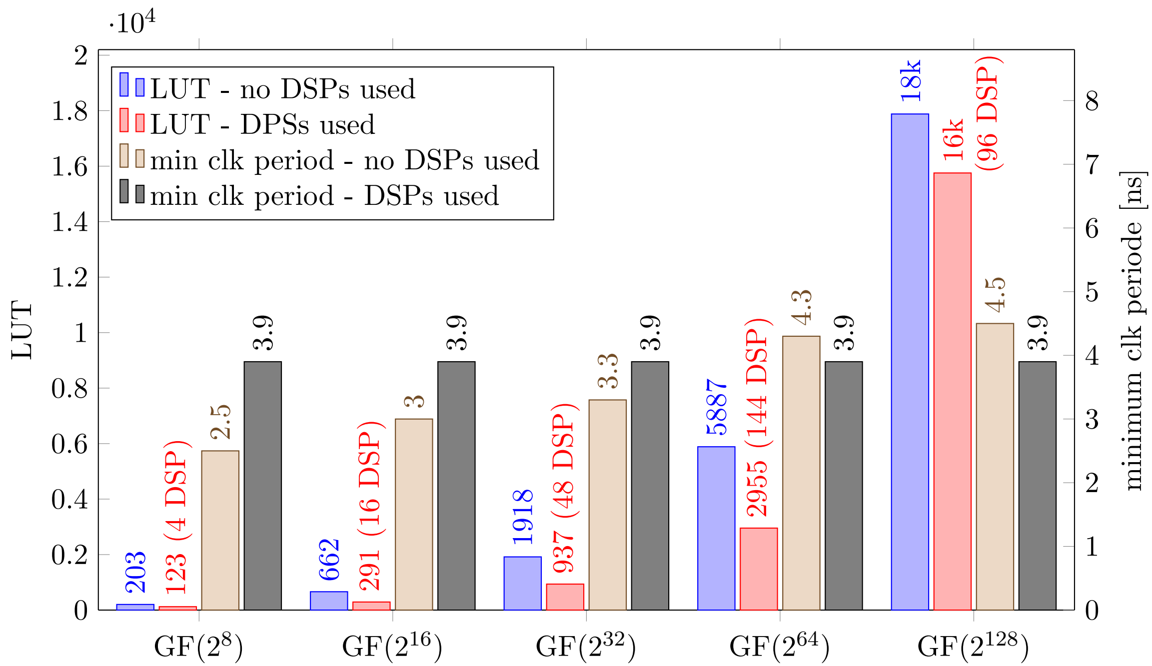


Figure 5.5: Size of the CSS SRU in 6-input LUT, for synthesis and implementation and the minimal possible clk period. It is shown for both, a pipelined and non-pipelined architecture.

In the following a share generation and secret reconstruction process is explained.

CSS - Share Generation

1. If a secret packet arrives, it is first stored in the Secret In Buffer.
2. The AES-Arbiter signalises the Secret In Buffer to send the data to the AES unit, if the Secret In Buffer contains a full secret packet, the SGU buffer is not full and the AES is free.
3. The header information is read if a new key is required. If so, a new key from the corresponding input *new_key* is read and stored, as well as the channels counter value is set to zero. Otherwise, the stored key is used and the counter continues with the previous value.
4. If a new key is used, the SGU buffers store the key with the packet it was applied for. The packet is sent to the SGU if the received packet is complete. If the packet contains a new key, it is handed over first to the SGU with the additional information to enable the SSS mode for the key-share. The rest of the data is processed in the ESS mode. At the same time the header information is passed to the Share Out Buffers.
5. The SGU simply receives the data and information which mode to use, produces the shares and sends them to the Share Out Buffers.
6. If a share buffer contains a whole packet, it signalises on the outer interface to have one share packet ready for pick up.

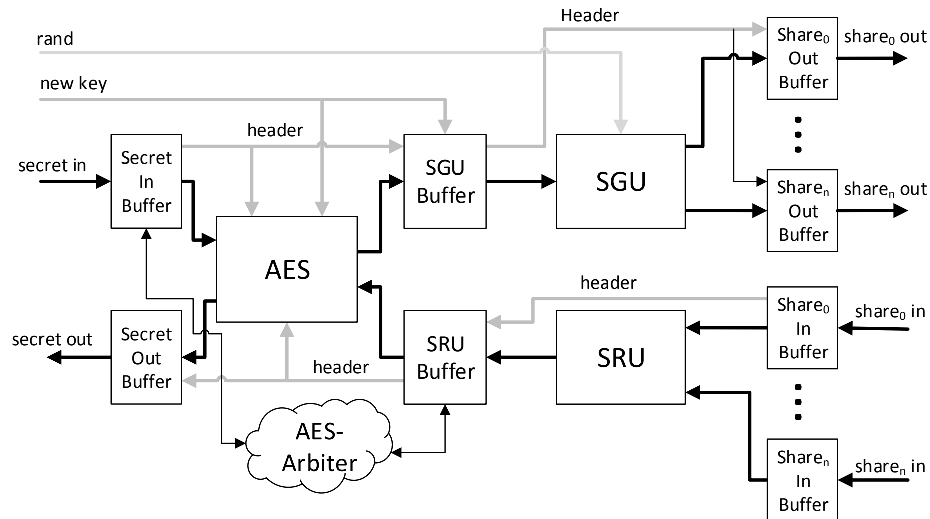


Figure 5.6: Simplified structure of the CSS unit.

CSS - Secret Reconstruction

1. If all the Share In Buffers contain a full packet and the SRU buffer is not full, the header information is passed on to the SRU buffer and the shares are handed to the SRU. If a key is to be extracted according to the header information, the SRU receives the share with the additional information to operate in the ESS mode. Afterwards the remaining data is sent in the SSS mode.
2. The SRU reconstructs the secret in the selected mode and hands the data to the SRU buffer.
3. The SRU uses the header information to determine whether the reconstructed data contains a key. If this is the case, the key is stored with its related packet.
4. If the SRU buffer contains a full packet, the AES free and the Secret Out Buffer is not full, the AES-Arbiter signalises the SRU buffer to send its data to the AES. The header is sent to the Secret Out Buffer as well as to the AES. If a new key was reconstructed, this key is also sent to the AES.
5. The AES checks if a new key was extracted and if so, resets the according counter value and uses the new key. Then all of the other data is decrypted.
6. If a secret packet is fully reconstructed, the Secret Out Buffer signalises the packet to be ready to the outer interface.

The buffer and packet size is generic in this design. However, similar sizes of the secret and share packets were chosen. Therefore k secret packets have to be loaded until the share packets are ready, where k is the threshold level of the applied n/k threshold scheme.

5.6 Evaluation

In accordance with the evaluation of Shamir's Secret Sharing scheme in chapter 3.4, a theoretical maximal throughput comparison between a software and a hardware implementation is given, as

well as an estimation for a real design.

Hardware Software Comparison

For the theoretical maximal throughput evaluation in a hardware implementation only the space consuming units AES, SGU and SRU were considered with the previous presented implementation results. The results were extrapolated for the assumption that 50% of the available LUTs on the Zynq®-7000 FPGA were utilized (50% = 26600 LUT). For the share generation the AES and SGU were considered, while for the share reconstruction the AES and SRU were taken into account. First AES, multiple SGU or SRU reached a total throughput of 128 bit per cycle. Then the resulting sizes were replicated until 26600 LUTs were utilized. The multiplier design was realized without DSPs in order to obtain a more objective comparison. In the software implementation, as well as in the hardware, all data flows were neglected. The library and multiplication strategies such as in chapter 3.4 were applied. It was extended by an AES function [36] which achieved a throughput of 156 Mbit/s with all four cores of the evaluation PC of section 3.4.

For both the hardware and software, 8 shares were generated and a threshold value of 4 was assumed. The final throughput results are presented in figure 5.7. The throughput is measured in Gbit/s for the output of the generated shares and reconstructed secrets for each functionality.

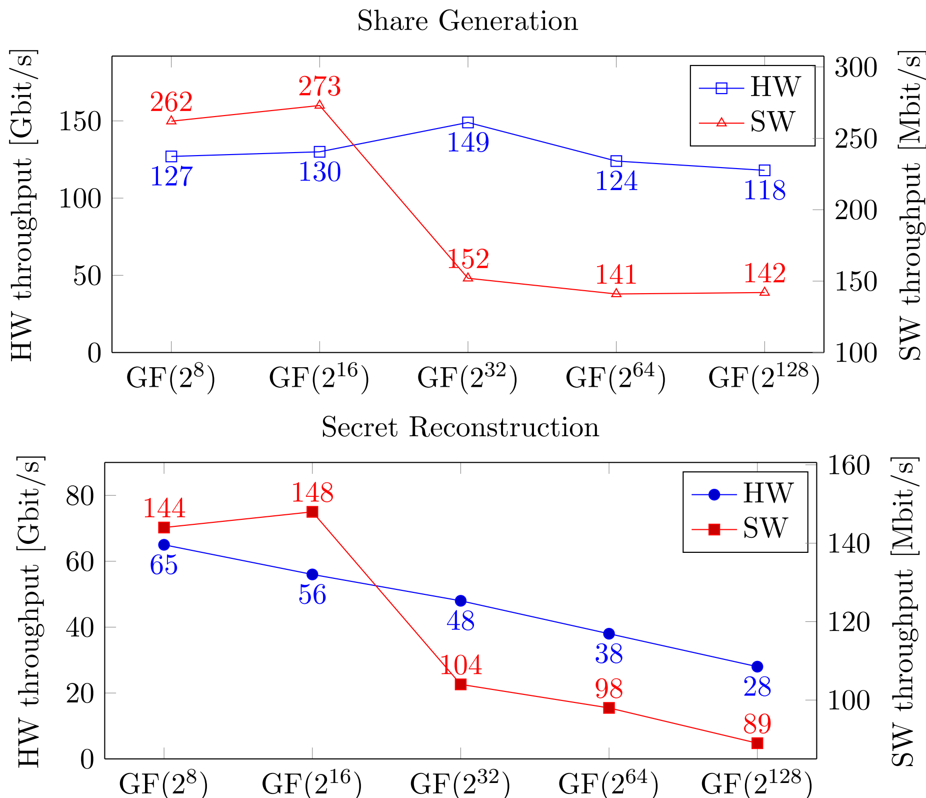


Figure 5.7: The theoretical maximal throughput of a software and a FPGA implementation respectively for the share generation and secret reconstruction. A parallel construction of 8 shares and a threshold value of 4 is assumed. The throughput is measured in share bits per second for the share generation and secret bits per second for the secret reconstruction.

In comparison to the previous implementation of Shamir's scheme, the influence of the Galois field decreases due to the influence of the AES, which remains the same in all Galois fields. A performance drop of a factor of about 2-3 for the hardware and 8-12 for the software implementation is observed. The hardware implementation remains significantly faster than its software counterpart.

Based on these values, a power estimation for a throughput of one Gbit/s were made. For the PC the CPU dissipation power of 84 W were used. The power consumption of the individual units (AES, SGU and SRU) was estimated in Vivado. The values were extrapolated for a theoretical 50% utilized FPGA.

A real implementation of all the units in parallel would lead to a lower power consumption. However, it can be seen as a reference value for an upper bound. The resulting values are presented in table 5.3.

Field	Share Generation		Secret Reconstruction	
	FPGA	PC	FPGA	PC
GF(2^8)	697mW	321W	57mW	585W
GF(2^{16})	404mW	308W	44mW	568W
GF(2^{32})	243mW	553W	35mW	808W
GF(2^{64})	188mW	598W	37mW	854W
GF(2^{128})	238mW	677W	38mW	984W

Table 5.3: Power consumptions for a throughput of 1 Gbit/s in the CSS scheme.

CSS - Core

The architecture of the CSS core shows the best performance at 64 bit. The share generation and secret reconstruction is performed fully parallel with an optimal utilization of the AES unit. Moreover, it is possible to implement a 64 bit system in a FPGA like the Zynq®-7000.

The implementation of a full Computational Secret Sharing system, integrated with a network connection is of high effort and presented in the next chapter 6. In table 5.4 a summary of the synthesis sizes of the full CSS core, including all buffers, is given. In the synthesized implementation 8 shares were generated parallel with 4 shares needed for the reconstruction. A packet size of 1500 bytes was selected, oriented at a maximal Ethernet packet size. Each buffer can store up to 2 packets and the header length was set to 16 bit.

	GF(2^{32})		GF(2^{64})		GF(2^{128})	
	LUT	BRAM	LUT	BRAM	LUT	BRAM
AES	8944	36	8946	36	8945	36
AES Arbiter	2		2		2	
Secret In Buffer	231	3	231	2	361	2
Secret Out Buffer	217	2.5	217	2.5	215	2.5
Share In Buffer	386	4	403	4	448	8
Share Out Buffer	1560	12	1808	12	1785	
SGU Buffer	304	2	335	2	334	2
SRU Buffer	320	2	317	2	243	2
SGU	887		1651		3638	20
SRU	936		2980		15743	
	+48 DSP		+144 DSP		+96 DSP	
Total	13791	60.5	16893	60.5	31717	72.5
	+48 DSP		+144 DSP		+96 DSP	

Table 5.4: Synthesis results of the CSS core.

6 A Full Computational Secret Sharing System

In this chapter the preciously introduced CSS core is applied to a complete system capable of sharing files. Additional elements to achieve this goal, such as a True Random Number Generator, a network communication and a protocol for the file management are presented.

6.1 Introduction

In a full applied secret sharing setup, a client wants to save a file on multiple servers. In between the client and the servers, the FPGA is settled. The client sends its data to the FPGA, which performs the secret sharing algorithm and splits a file into multiple shares, which are distributed to the servers. On the other hand, the FPGA is capable of putting shares together to reconstruct the original file. This process is invisible to the client. Such a setup is illustrated in figure 6.1.

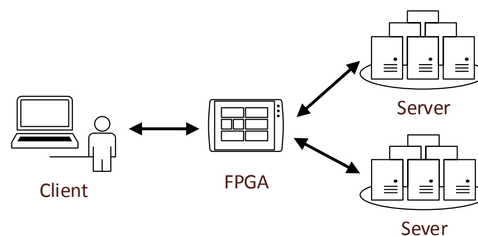


Figure 6.1: The secret sharing setup.

In order to establish a full secret sharing system, more components besides the CSS core are required. These components are identified in the following and discussed in the next sections of this chapter.

- **TRNG:** A TRNG delivers random data to the CSS core and enables a correct and secure functionality.
- **Protocol:** A protocol manages files, including their shares and fragments them into packets.

- **External Communication:** An interface is responsible for the external communication of the FPGA board to the client and servers and should be able to operate at a reasonable speed.
- **CSS core Wrapper:** In accordance to the external interface, the CSS core is wrapped or a bridge is implemented to enable the communication of the CSS core with the external interface within the FPGA.
- **Client:** The client and its interface are relevant for selecting files for sharing, reconstruction and sending these to the FPGA. Moreover, control information should be exchanged.
- **Server:** The servers are capable of storing multiple shares, or rather share packets, and to communicate with the FPGA.

The applied CSS core architecture was synthesized for a 8/4 threshold scheme and the secret sharing is performed in $GF(2^{64})$. It allows a parallel share generation and secret reconstruction with maximum throughput, due to the optimal sharing of the AES core in between the SGU and SRU. Moreover, 64 bit is the highest bit-width and therefore offers the most efficient implementation on the target FPGA.

6.2 True Random Number Generator

Although a FPGA should be a fully deterministic system, various works showed the possibility to generate true random numbers within it, e.g. [51], [72], [66], [32], [73]. While the randomness source is usually any kind of noise, in [73] three ways to extract such a noise in a FPGA are presented.

- Metastability: In a digital circuit, only the states low and high with defined voltage ranges are allowed. If a signal changes from one state to the other, it passes through a forbidden state. If the forbidden state is sampled with a FF, the FF could reach one of the two allowed logic states, or become metastable [58]. Due to noise inside the FF the circuits behaviour becomes totally stochastic [74].
- Chaos: Chaos can only by be extracted by analog components, such as quantization errors of analog to digital converters (ADC). However, because many FPGAs contain such a ADC, e.g. the Zynq 7000, this source may be available.
- Jitter: In simple words, jitter is the unpredictable deviation in a signals propagation delay. It is the most used macro effect in FPGA based TRNGs, e.g. by sampling a noisy clock signal near of its edges [73].

A couple of well known TRNG designs exist in literature. In the following, designs considered as important are outlined, while all of them claim to produce totally random bit streams.

The first design of a TRNG, mainly targeted for a FPGA, is introduced in [33]. It samples the jitter of a phase locked loop (PLL). Therefore the PLL output is sampled near the transition zones, influenced by the jitter using a rationally related clock of a higher frequency. Built-in-PLLs of the FPGA were used, limiting the design to the FPGAs containing PLLs.

This idea was adapted in [44], where ring oscillators (RO) are replace the PLLs. Two identical ROs were built in one slice of a Xilinx FPGA, where the place-and-route is performed manually in order to reach a similar behaviour of the ROs.

The design proposed in [37] works with Galois LFSR and Fibonacci LFSR, where the shift registers are replaced by inverters. Due to slight variations in the delays, the output can not be predicted. Both LFSRs are XOR-combined.

A design based on open-delay-changes was introduced in [20], which aims to violate the timing constraints of a D-FF. Therefore multiple D-FFs are in parallel and their inputs are slightly delayed, reachable by wiring. Moreover, instead of built-in-D-FFs discrete latches are built with LUTs due to the high metastability avoidance of modern D-FFs.

The design of Snunar et al. in [69] bases on multiple ROs combined via XOR-connection sampled with a predefined frequency. Each RO has an independent jitter and there is a determinable possibility to capture the RO output signal within this jitter. By using a sufficient amount of ROs, one RO is always captured within its jitter area and the output signal becomes random.

There are advantages and disadvantages inherent in every design, so in [31] the TRNGs were compared with various metrics. All of the investigated practical TRNG designs revealed weaknesses and therefore the selection of a TRNG has to follow the designs criteria.

The proposed architecture from Sunar was favoured, because its high possible data rate of about 100 Mbits per second and its easy feasibility as it does not require PLLs nor manual routing. In further investigations in [23] the design showed weaknesses mainly due to the high fan-in at the XOR-gate, combining the outputs of all ROs. In [76] Wold and Tan made improvements to overcome this problem, by using one dedicated D-FF after each RO. Moreover, they showed that no post-processing of the output is necessary in order to achieve results, if sufficient ROs are utilized. In this composition one and zero are equally likely in the output stream. The design passed the statistical tests NIST [59] and Diehard [52] for random number generation evaluations, if 25 or more ROs are employed with a sample frequency of 100 MHz.

The design for the TRNG of Wold and Tan found application in this work and is shown in figure 6.2. The number of inverters in one RO and the total count of ROs is generically parametrizable. It gives the possibility to adapt the design to the needs of security and size.

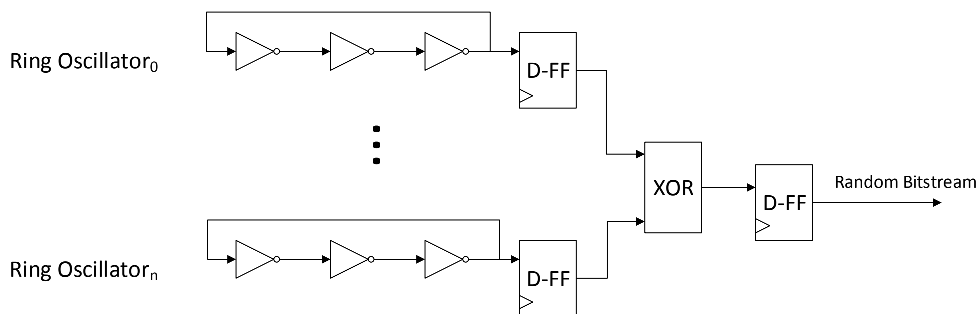


Figure 6.2: TRNG.

The required amount of random bits rely on the applied threshold scheme, the size of the key, the bit-width for the SGU, the packet size, how often a new key is used and the overall operating clock frequency of the CSS unit. Assuming a generation of a new key for every share packet, the

required amount of random bits N_r per second can be calculated as follows. First, the random bits per share packet are calculated as in equation (6.1), where N_k is the key size in bits, k the threshold value and bw the bit-with of the SGU. These bits are required periodically in a certain interval t, which is calculated in equation (6.2), using the quantity of secret packets with an individual key Q_p, the packet size in bits Np and the clock period $1/f_{clk}$. Finally, the required random bits per second is defined as in equation (6.3).

$$N_r = N_k + (k-1) \times bw \tag{6.1}$$

$$t = Q_p * \frac{N_p}{bw} \times \frac{1}{f_{clk}} \tag{6.2}$$

$$N_r/s = \frac{N_r}{t} \tag{6.3}$$

In a secret sharing scheme random bits are required at a rate of 42 Mbits/s with a threshold of 4, a packet size of 1500 bytes, a 64 bit SGU and a clock size of 100 MHz.

The applied TRNG produces random bits with a rate of 100 Mbit/s and is sufficient according to the estimation above. However, if configurations may require a higher rate, multiple random bit streams are generically produced in parallel in the final TRNG design.

6.3 Protocol

A protocol for the administration of complete files was developed. It takes care of identifying files, partitioning them into packets for transmissions in packet oriented networks, identifying packets and exchanging control information in between the client, FPGA board and servers. A lightweight protocol was developed and implemented to perform those tasks. Each packet is attached by a 128 bit header of the protocol, referred to as CSS header. The full capability of this protocol is explained by its fields. Table 6.1 shows the structure of the header, while the fields are explained in the following.

0 - 7	8 - 15	16 - 23	24 - 31
Padding	x-value	Command	Version
File Identifier			
Total Packet Count			
Sequence Number			

Table 6.1: Structure of the CSS protocol.

- **Version:** This field contains the header version number, intended for further extensions. For this work, only a version 0 was implemented.
- **Command:** This field determines the content of the packet. Following commands are possible:

- 0x01: A secret packet with the request to apply a new key. This is used for the first and then for every k secret packet.
- 0x02: A follower secret packet, to continue the encryption with the same key as for the secret packets before.
- 0x10: A share packet.
- 0x80: A request to return shares according to its identifier field. If this command is sent from the client, which must contain the numbers of the servers to restore the secret from. These numbers are attached in the payload after the header.
- 0x81: A heartbeat packet to the FPGA board. It can be used if a periodic transmission is necessary to keep the connection between the client and the FPGA board open, to refresh the Address Resolution Protocol (ARP) table or for debugging.
- 0x82: A heartbeat packet to the board, which is directly forwarded to all servers. The payload must contain the numbers of the servers to which the heartbeat should be forwarded. These numbers are attached in the payload after the header. It is used if a periodic transmission is required on all connections to keep a connection open, refresh the ARP table or for debugging.
- 0x90: A control packet, to set the Internet Protocol (IP_2) addresses for the three used Ethernet ports of the FPGA extension board. The IP_2 addresses are transmitted in the payload after the header.
- 0x91: A control packet for the IP_2 addresses of the servers. All packets belonging to servers are sent to these IP_2 addresses.
- 0xFE: A control packet for a forced reset for the FPGA logic.

- **x-value:** If the header is attached to a share packet, the field determines the x-value and distinguishes between the different share packets.
- **Padding:** This field is to maintain the header structure of 32 bit blocks.
- **Identifier:** A 32 bit identifier for a file. All packets according to the same file are carrying the same identifier. If the header is used for a request of a file or its shares, this field determines the returned shares.
- **Total Packet Count:** This field determines the total amount of packets for one file. With a packet payload size of 1424 byte a maximum file size of about 6 TB is supported.
- **Sequence Number:** This field determines the packet number within the total packet count.

6.4 External Communication

In order to communicate with the client and the servers, an external connection of reasonable speed is required. For this task Gigabit Ethernet was selected, including the Internet protocol and User Datagram Protocol (UDP) in order to communicate with a computer application. On top of UDP the CSS protocol is settled, which handles CSS specific tasks.

While the CSS core would be able to work at higher rates, Ethernet was chosen because of its widespread usage, which enables an easy integration of the FPGA system in a client-server environment. The Zedboard contains one Ethernet interface and was extended by a Quad Gigabit Ethernet FPGA Mezzanine Card, referred to as Ethernet FMC® [27]. The Ethernett FMC contains a total of four Ethernet ports, directly connected to the PL. One of those ports is dedicated to the client connection of which two are for the connection of the servers. For an 8/4 threshold scheme, a maximum utilization of all three channels is enabled in the share generation process. A theoretical throughput of 1 Gbit/s for secrets and 2 Gbit/s for shares was attained.

The extension board communicates with the PL via a reduced Gigabit Media Independent Interface (RGMII). Three AXI Ethernet IP_1 cores (AXI 1G/2.5G Ethernet Subsystem (7.0) [78] from Xilinx were applied in order to facilitate the communication with the Ethernet FMC. Furthermore, it handles basic functionalities of the Media Access (MAC) Layer, such as MAC address filtering and CRC calculation including CRC concatenation.

The IP_1 cores contain an AXI interface for setup information, connected to the PS. For the data transfer, the IP_1 core consist of one AXI interface for the data itself and one more for control sequences, respectively for receiving and transmitting.

While the IP_1 core performs the CR concatenation and MAC address filtering, other MAC layer tasks as well as the network and transmission layer responsibilities have to be handled by the overall system.

The header generation of the MAC, IP_2 and UPD differs for the secret and each share. Therefore the output buffers of the shares and the secret within the CSS core were extended to store these headers. After the header generation in the PS, they are loaded into the core. Every time the buffer puts out a packet, it first sends the MAC, IP_2 and UDP header, followed by the CSS header and then the actual payload.

With the selection of Ethernet, the packet size, including the IP_2, UDP and CSS headers, was set to 1500 bytes according to the maximum transfer unit of the Ethernet protocol [1].

6.5 Computational Secret Sharing Core Wrapper

A wrapper adapts the outer interfaces of the CSS core to the needs of the three Ethernet IP_1 cores. The outputs of the CSS core accords to the amount of generated shares and to the amount of shares needed for the reconstruction. Each input and output are matched to one of the three physical Ethernet interfaces. Figure 6.3 contain the architecture of the CSS core wrapper.

The secret input and output are connected to the Ethernet channel 0 (Ethernet IP_1 core 0). Additionally, a packet multiplexer and a packet switch are interposed to allow the PS to receive and send packets via Ethernet channel 0. It is determined from the CSS header information if the packet is forwarded to the CSS core or PS.

The share outputs are connected via the packet multiplexer to the Ethernet channel 0 and 1, four share outputs respectively. Additionally, a packet from the PS is always inserted. The reception and distribution of share packets for a reconstruction process is more complicated, as the packets can come from any of the two inputs. A share packet crossbar with both Ethernet channels as inputs sorts those packets according to their CSS header information, and delivers it to the corresponding CSS core input. The connection information, which share to forward on

which CSS core input, is obtained from a routing table, given from the PS. The PS calculates the elements of the inverse matrix for the reconstruction process and feeds them into the matching position of the CSS core. A branch from the share crossbar to the PS allows the reception of channel 0 and 1 packets for the PS.

All communication paths to the PS are grouped and made external via one AXI interface. The communication to the Ethernet ports are translated via Ethernet IP_1 core interfaces to the two necessary AXI channels.

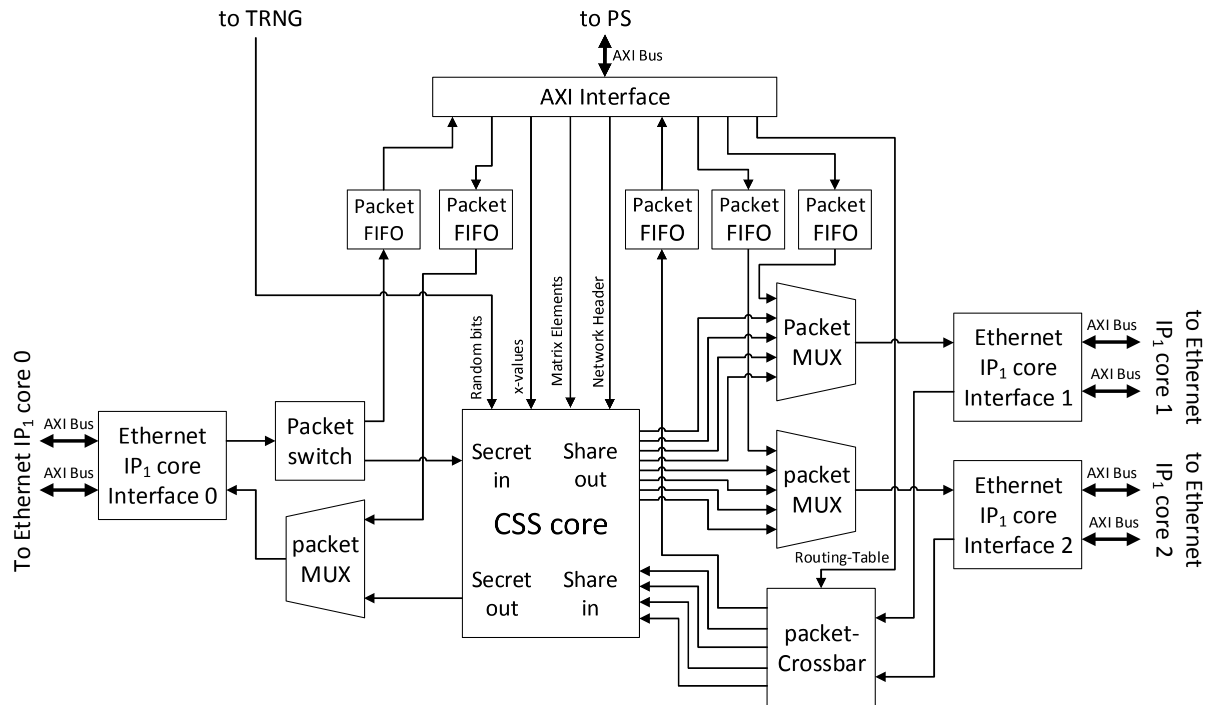


Figure 6.3: CSS core wrapper architecture.

6.6 Client And Server

The client and server are realized via computer applications in purpose of the evaluation setup. It enables a detailed analysis of stored packets. Both programs are written in Java. In the clients Graphical User Interface (GUI) additional parameters can be set. They include the x-values for the share generation, selecting servers for the secret reconstruction, the IP_2 addresses for the board as well as the IP_2 addresses for the servers and the client. For both applications the possibility to restrict the data rate is given in order to avoid congestion at the underlying layer or network.

The clients GUI allows to select files from the file system. These files are partitioned for the Ethernet transmission and a CSS header is attached. The final packets are sent to the FPGA board.

The server applications store share packets and return them if requested. Additional content is visualized graphically for an optical pattern detection.

6.7 Full Setup And Evaluation

In the final FPGA design, the CSS wrapper and TRNG were implemented as described above, one AXI interconnect, three AXI Ethernet IP_1 cores and a core for the organic light-emitting diode (OLED) display of the Zedboard. The display serves for status information, the saved IP_2 and MAC address of the board and client and servers for debug reasons. The client and server applications are performed on one computer. In the following, the process of the setup, share generation and reconstruction of the secret is described.

Configuration

For the configuration of the FPGA board, the client transmits all the IP_2 addresses of the servers, the client and the FPGA board itself to the board. This is performed by a broadcast message on the UPD port 16500. The board starts to send ARP requests and collects ARP responses in order to complete its ARP table for the later network communications.

Share Generation

The client application determines which x-values are included in the share generation process and transfers them to the FPGA. The PS of the FPGA sets the x-values in the according registers of the SGU and computes the MAC, IP_2 and UDP headers. After the configuration message, secret packets are sent, while k secret packets are needed to produce one share packet in a n/k threshold scheme. Afterwards, each generated share package is sent to the server.

Secret (-file) Reconstruction

The client application sends the information of which file is requested for the reconstruction process from which servers to the FPGA. The PS in the FPGA sends requests to the according servers. In the meantime the inverted matrix according to the requested servers and therefore x-values are calculated and sent in the PL. The routing table for the share packet crossbar is determined and sent to the PL. The secret is reconstructed, when all shares of the same packet number are received and sent to the client.

Utilization

The utilization of the FPGA is summarized in table 6.2. A considerable high amount of LUTs is required by the AXI Ethernet IP_1 cores. All the buffers consume a high amount of LUTs, on one hand due to their high occurrence and on the other hand due to their additional functionality of storing the CSS header and partly the MAC, IP_2 and UDP headers.

The placement of the design elements within the FPGA is demonstrated in figure 6.4, colour differentiated between the functional units.

Functional Unit	LUT		FF		BRAM	DSP
Full FPGA	53200	(100%)	106400	(100%)	140	220
Full Implementation	30495	(57%)	36273	(34%)	139	144
One Ethernet IP_1 core (0)	2708	(5.1%)	3984	(3.7%)	4	0
TRNG	108	(0.2%)	26	(0%)		0
OLED	2301	(4.3%)	1388	(1.3%)	1	0
Full CSS architecture	19144	(36%)	22158	(21%)	126	144
CSS Core	17382	(33%)	20452	(19%)	126	144
AES	3594	(6.8%)	8403	(7.9%)	100	0
SRU	3092	(5.8%)	7595	(7.1%)		144
SGU	2185	(4.1%)	1095	(1%)	0	0
Share Switch	323	(0.6%)	261	(0.2%)	0	0
All Buffers in CSS Core	8507	(16%)	3357	(3.2%)	26	0

Table 6.2: FPGA utilization for a complete CSS system as well as selected functional unit. THE SGU and SRU were implemented for 64 bit and a 4/8 threshold scheme was applied.

Evaluation

A complete network CSS system was successfully developed. Due to its dynamic IP_2 settings and protocols it is applicable in any Ethernet network. The CSS protocol enables the partitioning and management of whole files in a file sharing system. In this setup, multiple and arbitrary files could be selected and shared as well as completely restored. The full core is implemented within a single clock domain of 100 MHz, which leads to an internal CSS core throughput of 6.4 Gbit/s in both directions. The external speed is restricted to a maximum of 1 Gbit/s due to the limitation of the Ethernet speed. The full implementation consumes about 57% of LUT resources. In the practical setup the client and servers were performed on a computer. The files could be shared and restored with an affective rate of 250 Mbit/s due to congestion.

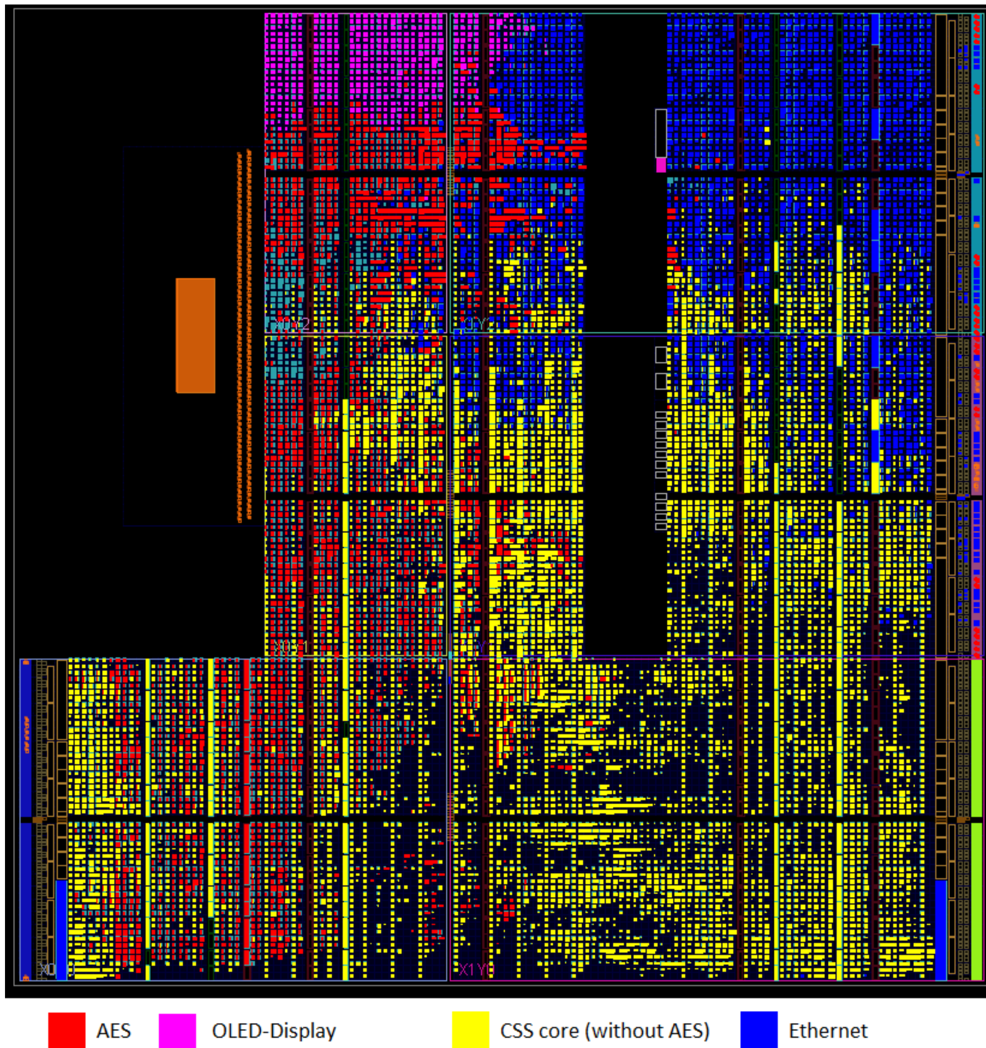


Figure 6.4: FPGA utilization

7 Conclusion

In this chapter the overall outcome of this work is summarized and discussed. Additional application fields for a secret sharing implementation are presented. Finally, improvement strategies and first estimations for a verified secret sharing are given, which can be the objective of future works.

7.1 Summary And Discussion

The implementation of Shamir's Secret Sharing scheme and the Computational Secret Sharing scheme in a FPGA was presented. The algorithms were analyzed in order to find an efficient implementation in terms of throughput. The multiplication was identified as the bottleneck of the system and improvements for the multiplier were discussed.

The implementation result were evaluated for the bit-widths of 8, 16, 32, 64 and 128 bit. A comparison to software implementations was obtained by evaluating the theoretical reachable maximum throughputs.

The result are generic IP_1 cores, parametrizeable for various parameters such as the bit-width, the amount of shares to be generated and the threshold value. Additionally, a full system was implemented which communicates with servers and a client using IP_2 over Ethernet, capable to share and distribute arbitrary files as well as restoring them.

This work shows a significant performance decrease for the application of wider bit-widths, as it is in software implementations. If the advantages for higher bit-widths are not used, the optimal approach would be to implement 8 bit implementations in parallel. However, the hardware implementation showed high performance gains compared to its software counterpart, by a significant reduction of power. Such a system is realizable in FPGAs and capable of operations at high data rates and even higher bit-widths.

For the SGU an almost constant performance in all of the investigated bit-widths is observable, due to the restriction of the possible x-values. The SRU shows almost a linear performance decrease, except for 32 bit, which only has a slight performance decrease compared to 16 bit.

The proposed architecture for a CSS implementation is best suited for 64 bit, due to the optimal utilization of all involved units. The final prototype approved a correct functionality in a completely integrated file based system.

7.2 Applications

The aim was to find an efficient hardware implementation, for applications in a cloud-of-cloud setup. As it is capable of a high throughput by a significant lower power consumption than a software counterpart, it is applicable for high data rates in data centers as well as for single users.

Due to its similar functionality it offers the usage as a secure Redundant Array of Independent Disks (RAID), specified in [4]. In a RAID system multiple hard discs are combined and redundancy is added in order to compensate a single fault of a hard disk. Newer RAID arrangements, such as RAID 6, are capable of handling two faulty disks. However, the proposed system is capable of handling an arbitrary number of faulty hard disks, selectable with the threshold value and the number of produced shares. Additionally, it offers more security than the RAID system, as the information is encrypted. If a hard disk is sorted out, their data do not need to be erased with high effort, as a single hard disk would never reveal any information and is useless for a possible attacker.

7.3 Further Work

In order to prepare the complete hardware implementation for a real world environment, it is of advantage to add more robustness to the system as there may exist cheaters. The concept of verified secret sharing handles these issues, first introduced in [14] and briefly explained in chapter 2.5.

Related to this work, first attempts were made to generate the required fingerprints with the secure hash algorithm (SHA) SHA-256 [3] or SHA-3 [6]. To compete with the share generation speed, all hash values are generated in parallel, with input rates from 8 - 128 bits per cycle. An analysis of applicable implementations lead to a certified SHA-3 core from Opencores [**?**]. With a throughput of 7.2 Gbit/s it competes with the SGU and SRU implementations. The core has a size of 9895 LUTs in a Virtex 6 FPGA (XC6VLX240T-1FF1156). Performing the hash generations in parallel for all shares would exceed the size capability of the target FPGA. In a FPGA of bigger size, the fingerprint generation unit would still take the biggest portion of the resources. Therefore the efficiency and optimized realizations for a fingerprint generation as well as a complete verified secret sharing implementation could be the objectives of further works.

Appendix

The appendix is structured as follows. In section A, first screen-shots of the final project in Vivado are presented followed by an additional block diagram of the CSS core. In section B a Java code snapshots is presented, which was used to generate the static XOR-connection for the reduction circuit, as well as example reduction circuits in VHDL. In section C, snapshots of important VHDL modules are shown. Finally the C code is presented in section D. The code for the matrix inversion is presented as well as the example code, to evaluate the maximal reachable secret sharing throughput on a PC.

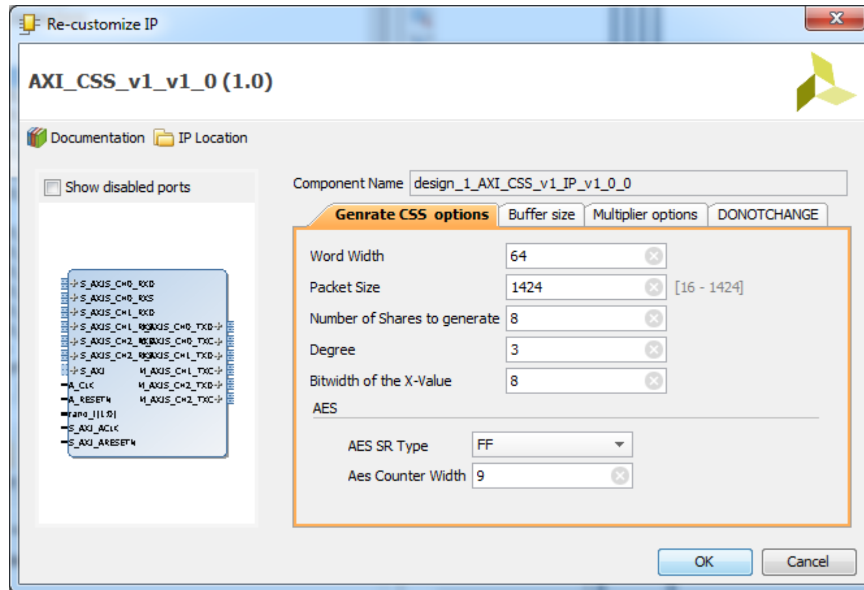
A - Vivado Screen-Shots and Block Diagrams

The Vivado block design of the full implementation

This image shows the block diagram of the full implementation, capable to apply secret sharing for complete files in a network environment. It contains the processing system, AXI- interconnects, the AXI-Ethernet IP_1 cores and the CSS core.

AXI CSS core - customization GUI

In this subsection a screen-shot of the developed AXI CSS core is given. In the table below all of the customization parameters are summarized.



General CSS Options	
Word Width:	bit-width of the Secret Sharing architecture
Packet Size:	size of secret and share packets in byte, typ. 1424 byte (payload) for Ethernet
Number of Shares to generate:	number of shares to generate parallel
Degree:	degree of the Shamir-polynomial $(k-1)$
Bit-width of the x-Value:	bit-width of the x-value

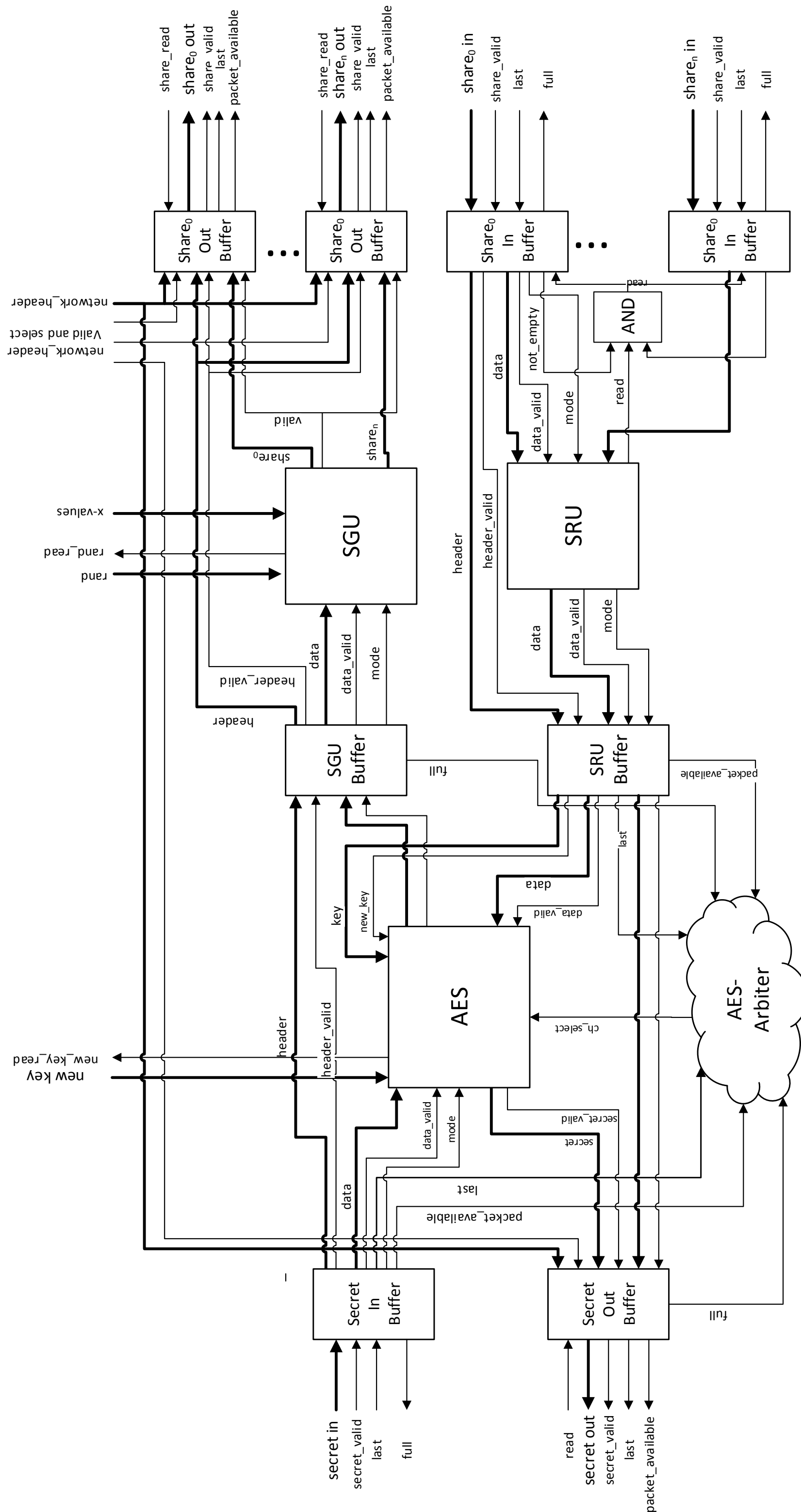
AES	
AES SR Type:	shift register type of the AES, BRAM of FF
AES Counter Width:	counter width of the counter in CTR mode

Buffer size	
Secret In Buffer Max Packets	
Share Out Buffer Max Packets	
Share In Buffer Max Packets	
Secret Out Buffer Max Packets	
SGU Buffer Max packets	
SRU Buffer Max packets	

Multiplier Options	
Pipelining:	Pipelining stages for each 16×16 sub-multiplier (0,1,2,3)
Usage DSP:	how many DSPs should be used in one 16×16 multiplier (0,1,2,3)
DSP Per Karatsuba	many of the three multipliers in the Karatsuba algorithm should be realized using DSPs

Detailed CSS core

This figure presents the detailed architecture of the CSS core. In comparison to figure 5.6 the Share Out Buffers and the Secret Out Buffer are extended by the capability to store and output network headers.



B - The Reduction Circuit

Calculation of the static XOR-connection for the reduction

A snapshot of the Java program, to determine the static XOR-connections for the reduction circuit, as described in [81].

```
static void CalcXORConnection(String polynom, int dataWidth,
  String inputSignalName, String inputSignalName){

  String Nin = new String();     //input String for CRC

  /**************** Matrix generation ********************/
  int[][] XOR_Table= new int[dataWidth][polynom.length()-1];
  int k=dataWidth;

  for(int i=0; i<dataWidth;i++){
    k=k-1;
    Nin=getStringZeros(k)+"1"+getStringZeros(i);
    XOR_Table[i] = String_to_Int_Array(CalcCRC(polynom, Nin));
  }

  /************** Generate XOR VHDL Codes ****************/
  boolean begin=true;
  int XORPathCnt=0;
  int XORLongestPath=0;

  for(int i=0;i<polynom.length()-1;i++){
    System.out.println();
    System.out.print(inputSignalName+"("+i+") <= ");
    for(int j=0;j<dataWidth;j++){

      if(XOR_Table[j][(polynom.length()-2)-i]==1){
        if(begin==true){
          begin=false;
        }else{
          System.out.print(" XOR ");
          XOR_path_cnt++;
        }
        System.out.print(""+inputSignalName+"("+j+")");
      }
    }
    if(XORPathCnt > XORLongestPath){ XORLongestPath=XORPathCnt;}
    XOR_path_cnt=0;
    System.out.print(";");
    begin=true;
  }
  System.out.println("Nr. of XORs in longest path: \t"+XORLongestPath);
  return;
}
```

In the following the resulting static XOR-connection for the binary extension fields GF(2^8), GF(2^{16}) and GF(2^{32}) applying the irreducible polynomials of table 3.4 are listed.

Static XOR-connection for GF(2^8)

```
r(0) <= d(0) XOR d(8)  XOR d(12) XOR d(13);
r(1) <= d(1) XOR d(8)  XOR d(9)  XOR d(12) XOR d(14);
r(2) <= d(2) XOR d(9)  XOR d(10) XOR d(13);
r(3) <= d(3) XOR d(8)  XOR d(10) XOR d(11) XOR d(12) XOR d(13) XOR d(14);
r(4) <= d(4) XOR d(8)  XOR d(9)  XOR d(11) XOR d(14);
r(5) <= d(5) XOR d(9)  XOR d(10) XOR d(12);
r(6) <= d(6) XOR d(10) XOR d(11) XOR d(13);
r(7) <= d(7) XOR d(11) XOR d(12) XOR d(14);
```

Static XOR-connection for GF(2^{16})

```
r(0)  <= d(0)  XOR d(16) XOR d(27) XOR d(29);
r(1)  <= d(1)  XOR d(16) XOR d(17) XOR d(27) XOR d(28) XOR d(29) XOR d(30);
r(2)  <= d(2)  XOR d(17) XOR d(18) XOR d(28) XOR d(29) XOR d(30);
r(3)  <= d(3)  XOR d(16) XOR d(18) XOR d(19) XOR d(27) XOR d(30);
r(4)  <= d(4)  XOR d(17) XOR d(19) XOR d(20) XOR d(28);
r(5)  <= d(5)  XOR d(16) XOR d(18) XOR d(20) XOR d(21) XOR d(27);
r(6)  <= d(6)  XOR d(17) XOR d(19) XOR d(21) XOR d(22) XOR d(28);
r(7)  <= d(7)  XOR d(18) XOR d(20) XOR d(22) XOR d(23) XOR d(29);
r(8)  <= d(8)  XOR d(19) XOR d(21) XOR d(23) XOR d(24) XOR d(30);
```

```
r(9)  <= d(9)  XOR d(20) XOR d(22) XOR d(24) XOR d(25);
r(10) <= d(10) XOR d(21) XOR d(23) XOR d(25) XOR d(26);
r(11) <= d(11) XOR d(22) XOR d(24) XOR d(26) XOR d(27);
r(12) <= d(12) XOR d(23) XOR d(25) XOR d(27) XOR d(28);
r(13) <= d(13) XOR d(24) XOR d(26) XOR d(28) XOR d(29);
r(14) <= d(14) XOR d(25) XOR d(27) XOR d(29) XOR d(30);
r(15) <= d(15) XOR d(26) XOR d(28) XOR d(30);
```

Static XOR-connection for GF(2^{32})

```
r(0)  <= d(0)  XOR d(32) XOR d(57) XOR d(61) XOR d(62);
r(1)  <= d(1)  XOR d(33) XOR d(58) XOR d(62);
r(2)  <= d(2)  XOR d(32) XOR d(34) XOR d(57) XOR d(59) XOR d(61) XOR d(62);
r(3)  <= d(3)  XOR d(32) XOR d(33) XOR d(35) XOR d(57) XOR d(58) XOR d(60) XOR d(61);
r(4)  <= d(4)  XOR d(33) XOR d(34) XOR d(36) XOR d(58) XOR d(59) XOR d(61) XOR d(62);
r(5)  <= d(5)  XOR d(34) XOR d(35) XOR d(37) XOR d(59) XOR d(60) XOR d(62);
r(6)  <= d(6)  XOR d(35) XOR d(36) XOR d(38) XOR d(60) XOR d(61);
r(7)  <= d(7)  XOR d(32) XOR d(36) XOR d(37) XOR d(39) XOR d(57);
r(8)  <= d(8)  XOR d(33) XOR d(37) XOR d(38) XOR d(40) XOR d(58);
r(9)  <= d(9)  XOR d(34) XOR d(38) XOR d(39) XOR d(41) XOR d(59);
r(10) <= d(10) XOR d(35) XOR d(39) XOR d(40) XOR d(42) XOR d(60);
r(11) <= d(11) XOR d(36) XOR d(40) XOR d(41) XOR d(43) XOR d(61);
r(12) <= d(12) XOR d(37) XOR d(41) XOR d(42) XOR d(44) XOR d(62);
r(13) <= d(13) XOR d(38) XOR d(42) XOR d(43) XOR d(45);
r(14) <= d(14) XOR d(39) XOR d(43) XOR d(44) XOR d(46);
r(15) <= d(15) XOR d(40) XOR d(44) XOR d(45) XOR d(47);
r(16) <= d(16) XOR d(41) XOR d(45) XOR d(46) XOR d(48);
r(17) <= d(17) XOR d(42) XOR d(46) XOR d(47) XOR d(49);
r(18) <= d(18) XOR d(43) XOR d(47) XOR d(48) XOR d(50);
r(19) <= d(19) XOR d(44) XOR d(48) XOR d(49) XOR d(51);
r(20) <= d(20) XOR d(45) XOR d(49) XOR d(50) XOR d(52);
r(21) <= d(21) XOR d(46) XOR d(50) XOR d(51) XOR d(53);
r(22) <= d(22) XOR d(47) XOR d(51) XOR d(52) XOR d(54);
r(23) <= d(23) XOR d(48) XOR d(52) XOR d(53) XOR d(55);
r(24) <= d(24) XOR d(49) XOR d(53) XOR d(54) XOR d(56);
r(25) <= d(25) XOR d(50) XOR d(54) XOR d(55) XOR d(57);
r(26) <= d(26) XOR d(51) XOR d(55) XOR d(56) XOR d(58);
r(27) <= d(27) XOR d(52) XOR d(56) XOR d(57) XOR d(59);
r(28) <= d(28) XOR d(53) XOR d(57) XOR d(58) XOR d(60);
r(29) <= d(29) XOR d(54) XOR d(58) XOR d(59) XOR d(61);
r(30) <= d(30) XOR d(55) XOR d(59) XOR d(60) XOR d(62);
r(31) <= d(31) XOR d(56) XOR d(60) XOR d(61);
```

C - VHDL Code Snapshots

In this section snapshots of important VHDL modules are listed.

Secret Reconstruction Unit (Computational Secret Sharing)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use work.polynom_multiplier_top_pkg.all;
use work.reduction_pkg.all;
use work.functions_pkg.all;

entity secret_reconstruction is
  generic (M                : integer := 8;
           SHARES           : integer := 4;
           DSPPERKARATSUBA  : integer := 2;  -- 0, 2 or 3
           USEDSP           : integer := 2;  -- o, 2, 3 or 4
           PIPELINING       : integer := 1;  --
           MATRIX_MODE      : string  := "parallel");
  Port (clk                 : in  std_logic;
        rst                 : in  std_logic;
        matrix_port_i       : in  std_logic_vector((SHARES*shares*M)-1 downto 0);
        matrix_coef_i       : in  std_logic_vector(M-1 downto 0);
        matrix_coef_valid_i : in  std_logic;
        coef_sel_share_i    : in  integer range 0 to SHARES-1;
        coef_sel_pos_i      : in  integer range 0 to SHARES-1;
        share_port_i        : in  std_logic_vector((SHARES*M)-1 downto 0);
        secret_o            : out std_logic_vector(M-1 downto 0);
        secret_valid_o      : out std_logic;
        share_valid_i       : in  std_logic_vector(SHARES-1 downto 0);
        css_mode_i                 : in  std_logic;
        css_mode_o              : out std_logic;
        ready_o                    : out std_logic );
end secret_reconstruction;

architecture Behavioral of secret_reconstruction is

  constant PIPELIE_STAGES  : integer := get_pipeline_stages(m, PIPELINING);

  type bus_array     is array(0 to SHARES-1) of std_logic_vector(M-1 downto 0);
  type bus_array_big is array(0 to SHARES-1) of std_logic_vector(2*(M-1) downto 0);
  type Matrix        is array(0 to SHARES-1, 0 to SHARES-1) of std_logic_vector(M-1 downto 0);

  signal coef              : Matrix;
  signal next_coef         : bus_array;
  signal share_in          : bus_array;
  signal next_share        : bus_array;
  signal mult_unreduced    : bus_array_big;
  signal mult_reduced      : bus_array;
  signal cnt               : integer range 0 to SHARES-1;

  signal set_secret_valid   : std_logic;
  signal secret_valid_shift : std_logic_vector(PIPELIE_STAGES downto 0);
  signal css_mode_shift     : std_logic_vector(PIPELIE_STAGES+1 downto 0);
  signal ready_internal         : std_logic;
  signal run_reconstruction : std_logic;

begin

    --write the whole matrix coeficents parallel into FFs
    --and connect the big share port to individual share signals
    matrix_parallel: if(MATRIX_MODE = "parallel") generate
        connect: for i in 0 to SHARES-1 generate
            matrix: for j in 0 to SHARES-1 generate
                coef(i, j) <=  matrix_port_i( (i+1)*M + j*SHARES*M -1 downto i*M + j*SHARES*M);
            end generate;
            share_in(i) <=  share_port_i( ((i+1)*M) -1 downto i*M);
        end generate;
    end generate;

    --write the matrix coeficents sequential into a memory
    --and connect the big share port to individual share signals
    matrix_sequential: if(MATRIX_MODE = "sequential") generate
        connect: for i in 0 to SHARES-1 generate
            share_in(i) <=  share_port_i( ((i+1)*M) -1 downto i*M);
        end generate;
        process(clk) is
        begin
            if(clk'event and clk='1') then
                if(rst = '1') then
                    coef <= (others => ( others => (others => '0')));
                elsif(matrix_coef_valid_i = '1') then
                    coef(coef_sel_share_i, coef_sel_pos_i) <=  matrix_coef_i;
                end if;
            end if;
```

```
        end process;
    end generate;


    --instatiate as many multiplayers and reduction units as shares exists
    Mult: for i in 0 to SHARES-1 generate
        mult: polynom_multiplier_top
            generic map(M                  => M,
                        DSPPERKARATSUBA => DSPPERKARATSUBA,
                        USEDSP             => USEDSP,
                        PIPELINING         => PIPELINING)
            port map(clk      => clk,
                     ain_i    => next_share(i),
                     bin_i    => next_coef(i),
                     result_o => mult_unreduced(i));

        reduct: Reduction
            generic map (M => M)
            port map(Data_in_i       => mult_unreduced(i),
                     Data_reduced_o => mult_reduced(i));
    end generate;


    --XOR all Multiplication results
    process(clk) is
        variable temp_secret : std_logic_vector(M-1 downto 0) := (others => '0');
    begin
        if(clk'event and clk='1') then
            temp_secret := (others => '0');
            for i in 0 to SHARES-1 loop
                temp_secret := temp_secret xor mult_reduced(i);
            end loop;
            secret_o <= temp_secret;
        end if;
    end process;


    --shift the valid and mode states
    process(clk) is
    begin
        if(clk'event and clk='1') then
            if rst = '1' then
                secret_valid_shift <= (others => '0');
                css_mode_shift <= (others => '0');
            else
                secret_valid_shift <= secret_valid_shift(PIPELIE_STAGES-1 downto 0) & set_secret_valid;
                css_mode_shift <= css_mode_shift(PIPELIE_STAGES downto 0) & css_mode_i;
            end if;
        end if;
    end process;

    --output if secrets are valid
    secret_valid_o <= secret_valid_shift(PIPELIE_STAGES);
    css_mode_o <= css_mode_shift(PIPELIE_STAGES+1);

    --load the correct coefficents and prepare the valid flag
    process(clk) is
    begin
        if(clk'event and clk='1') then
            if(rst='1') then
                cnt <= 0;
                run_reconstruction <= '0';
                set_secret_valid <= '0';
                for i in 0 to SHARES-1 loop
                    next_share(i) <= (others => '0');
                end loop;
            elsif share_valid_i(0) = '1' or run_reconstruction = '1' then
                run_reconstruction <= '1';
                cnt <= cnt + 1;

                --load the next shares
                if cnt = 0 then
                    for i in 0 to SHARES-1 loop
                        next_share(i) <= share_in(i);
                    end loop;
                end if;

                --select and load the next coefficients
                for i in 0 to SHARES-1 loop
                    next_coef(i) <= coef(i, cnt);
                end loop;

                --prepare the secret valid flag
                set_secret_valid <= '1';

                --All secret reconstructed
                if((css_mode_i = '1' and cnt = SHARES-1) or css_mode_i = '0') then
                    cnt <= 0;
                    run_reconstruction <= '0';
                end if;
            else
                set_secret_valid <= '0';
```

```
            end if;
        end if;
    end process;


    --decide if the calculation is finished and new values should be request
    process(share_valid_i(0), css_mode_i, cnt, rst) is
    begin
        if(rst='1') then
            ready_internal <= '1';
        elsif(css_mode_i = '0') then
            ready_internal <= '1';
        elsif cnt = SHARES-1 then
            ready_internal <= '1';
        elsif cnt = 0 and share_valid_i(0) = '0' then
            ready_internal <= '1';
        elsif cnt = 0 and share_valid_i(0) = '1' then
            ready_internal <= '0';
        else
            ready_internal <= '0';
        end if;
    end process;

    ready_o <= ready_internal;

end Behavioral;
```

Share Generation Unit (Computational Secret Sharing)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.polynom_multiplier_pkg.all;
use work.reduction_pkg.all;

entity sharegen is
  generic (M          : integer := 32;
           x_BW       : integer := 8;
           DEGREE     : integer := 3;
           SHARE_NUM : integer := 8 );
  port (clk              : in  std_logic;
        rst              : in  std_logic;
        secret_i         : in  std_logic_vector (m-1 downto 0);
        x_port_i         : in  std_logic_vector((SHARE_NUM*x_BW)-1 downto 0);
        data_in_valid_i : in  std_logic;
        shares_o         : out std_logic_vector((SHARE_NUM*M)-1 downto 0);
        shares_valid_o  : out std_logic;
        rand_i           : in  std_logic_vector(M-1 downto 0);
        rd_rand_o        : out std_logic;
        mode_i           : in  std_logic;
        rd_data_o        : out std_logic );
end sharegen;

architecture Behavioral of sharegen is

    type bus_array is array(0 to Share_num-1) of std_logic_vector(M-1 downto 0);
    type bus_array_x is array(0 to Share_num-1) of std_logic_vector(X_BW-1 downto 0);
    type bus_array_unreduced is array(0 to Share_num-1) of std_logic_vector(2*(M-1) downto 0);

    constant zeros         : std_logic_vector((M-x_BW)-1 downto 0) := (others => '0');
    signal x               : bus_array;
    signal share           : bus_array;
    signal factor          : bus_array;
    signal mult_unreduced : bus_array_unreduced;
    signal mult_reduced   : bus_array;
    signal temp_share     : bus_array;
    signal summand_buf    : std_logic_vector(M-1 downto 0);
    signal cnt             : integer range 0 to degree;
    signal secret_buf     : std_logic_vector(M-1 downto 0);
    signal share_valid_sr : std_logic_vector(1 downto 0);
    signal css_mode        : std_logic;
    signal rd_data         : std_logic;
    signal start           : std_logic;

begin

    connect: for i in 0 to SHARE_NUM-1 generate
        x(i) <= zeros & x_port_i( ((i+1)*X_BW) -1 downto i*X_BW);
        shares_o( ((i+1)*M) -1 downto i*M) <= share(i);
    end generate;

    myPEU: for i in 0 to share_num-1 generate
        mult_s: polynom_multiplier generic map(M => M)
            port map(clk    => '0',
                     a      => x(i),
                     b      => factor(i),
                     result => mult_unreduced(i) );

        reduct: Reduction generic map (M => M)
            port map(Data_in_i       => mult_unreduced(i),
                     Data_reduced_o => mult_reduced(i) );

        temp_share(i) <= summand_buf xor mult_reduced(i);
    end generate;

    shares_valid_o <= share_valid_sr(1);
    start <= data_in_valid_i or (not css_mode);

    --calculate the share-values
    process(clk)
    begin
        if (clk'event and clk = '1') then
            rd_rand_o <= '0';
            share_valid_sr(0) <= '0';
            share_valid_sr(1) <= share_valid_sr(0);
            share <= temp_share;
            if(rst = '1') then
                rd_rand_o <= '0';
                cnt <= 0;
                share_valid_sr <= "00";
            elsif(start='1') then
                if css_mode = '0' then
                    summand_buf <= rand_i;
                    rd_rand_o <= '1';
                else
                    summand_buf <= secret_i;
                end if;

                for i in 0 to Share_num-1 loop
```

```
                    factor(i) <= temp_share(i);
                end loop;

                cnt <= cnt+1;
                if(cnt = 0) then
                    for i in 0 to Share_num-1 loop
                        factor(i) <=  (others => '0');
                    end loop;
                elsif(cnt = DEGREE) then
                    cnt <= 0;
                    share_valid_sr(0) <= '1';
                end if;
            end if;
        end if;
    end process;

    --read the next data according to thee correct mode
    process(clk)
    begin
        if (clk'event and clk = '1') then
            if(rst = '1') then
                css_mode    <= '1';
                rd_data <= '1';
            elsif(start='1') then
                if mode_i = '0' or css_mode = '0' then
                    css_mode <= '0';
                    rd_data <= '0';
                end if;

                if(cnt = DEGREE-1) then
                    rd_data <= '1';
                elsif(cnt = DEGREE) then
                    css_mode <= '1';
                    rd_data <= '1';
                end if;
            end if;
        end if;
    end process;

                -- signalize if data is read
    process(rd_data, mode_i, cnt, start) is
    begin
        if(rd_data = '1') then
            if (mode_i = '0' and start = '1') and cnt = 0 then
                rd_data_o <= '0';
            else
                rd_data_o <= '1';
            end if;
        else
            rd_data_o <= '0';
        end if;
    end process;
end Behavioral;
```

CSS core

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.functions_pkg.all;
use work.packet_in_buffer_pkg.all;
use work.AES_pkg.all;
use work.AES_channel_arbiter_pkg.all;
use work.secrec_out_buffer_pkg.all;
use work.sharegen_in_buffer_pkg.all;
use work.sharegen_pkg.all;
use work.out_buffer_pkg.all;
use work.share_in_buffer_pkg.all;
use work.secret_reconstruction_pkg.all;
use work.secret_in_buffer_pkg.all;

entity CSS_v1 is
    generic(M                               : integer := 64;
            x_BW                            : integer := 8;
            Share_num                       : integer := 12;
            degree                          : integer := 6;
            DSPperKaratsuba                 : integer := 3;
            UseDSP                          : integer := 3;
            Pipelining                      : integer := 2;
            MATRIX_MODE                     : string  := "parallel";
            EXTERNAL_BUS_WIDTH              : integer := 32;
            PACKET_SIZE                     : integer := 32;
            header_size                     : integer := 16;
            SECRET_IN_BUFFER_MAX_PACKETS    : integer := 2;
            SHARE_OUT_BUFFER_MAX_PACKETS    : integer := 2;
            SECRET_OUT_BUFFER_MAX_PACKETS   : integer := 2;
            SHARE_IN_BUFFER_MAX_PACKETS     : integer := 2;
            SHAREGEN_BUFFER_MAX_PACKETS     : integer := 2;
            SECREC_BUFFER_MAX_PACKETS       : integer := 2;
            AES_SR_TYPE                     : string  := "BRAM";
            AES_COUNTER_WIDTH               : integer := 16 );
    port(clk                  : in  std_logic;
         rst                  : in  std_logic;
         CSS_Ctrl             : in  std_logic_vector(32 - 1 downto 0);
         rand                 : in std_logic_vector(M-1 downto 0);
         rand_rd_o            : out std_logic;
         --Share generator

         x_port               : in  std_logic_vector((share_num * x_bw) - 1 downto 0);
         secret_in            : in  std_logic_vector(EXTERNAL_BUS_WIDTH - 1 downto 0);
         secret_in_write      : in  std_logic;
         secret_in_last       : in std_logic;
         share_out_port       : out std_logic_vector((SHARE_NUM * EXTERNAL_BUS_WIDTH) - 1 downto 0);
         ShareGen_Status      : out std_logic_vector(32 - 1 downto 0);

         ShareGen_Buff_rd_en         : in std_logic_vector (Share_num - 1 downto 0);
         ShareGen_Buff_rd_valid      : out std_logic_vector(SHARE_NUM - 1 downto 0);
         ShareGen_Buff_packet_ready  : out std_logic_vector(SHARE_NUM - 1 downto 0);
         ShareGen_Buff_last          : out std_logic_vector(SHARE_NUM - 1 downto 0);

         nt_header_i          : in std_logic_vector(31 downto 0);
         sel_nt_header_i      : in std_logic_vector(SHARE_NUM - 1 + 1 downto 0);
         wr_nt_header_i       : in std_logic;
         rst_nt_header_cnt_i  : in std_logic;

         --Secret reconstruction
         matrix_port          : in  std_logic_vector(((DEGREE+1)*(DEGREE+1)*M)-1 downto 0);
         matrix_coef_i        : in std_logic_vector(M-1 downto 0);
         matrix_coef_valid_i  : in std_logic;
         coef_sel_share_i     : in integer range 0 to SHARE_NUM-1;
         coef_sel_pos_i       : in integer range 0 to SHARE_NUM-1;
         share_in_port        : in  std_logic_vector(((DEGREE+1)*EXTERNAL_BUS_WIDTH)-1 downto 0);
         share_in_write       : in  std_logic_vector((DEGREE+1)-1 downto 0);
         secret_out           : out std_logic_vector(EXTERNAL_BUS_WIDTH-1 downto 0);
         secret_out_available : out std_logic;
         secret_out_read      : in  std_logic;
         secret_out_valid     : out std_logic;
         secret_out_last      : out std_logic;
         SecRec_Status        : out std_logic_vector(32 - 1 downto 0);
         new_key              : in std_logic_vector(128-1 downto 0);
         new_key_rd_o         : out std_logic );
end CSS_v1;

architecture Behavioral of CSS_v1 is

    type share_in_bus_array      is array(0 to DEGREE) of std_logic_vector(EXTERNAL_BUS_WIDTH-1 downto 0);
    type share_in_buf_bus_array  is array(0 to DEGREE) of std_logic_vector(M-1 downto 0);
    type bus_array               is array(0 to Share_num-1) of std_logic_vector(M-1 downto 0);
    type share_out_bus_array     is array(0 to Share_num-1) of
                                    std_logic_vector(EXTERNAL_BUS_WIDTH-1 downto 0);
    type share_in_header_array   is array(0 to DEGREE) of std_logic_vector(128-1 downto 0);


    signal secret_in_buffer_out_last          : std_logic;
    signal secret_in_buffer_out_data          : std_logic_vector(128-1 downto 0);
    signal secret_in_buffer_heade_out_valid   : std_logic;
```

```
    signal secret_in_buffer_read             : std_logic;
    signal secret_in_buffer_packet_available : std_logic;
    signal secret_in_buffer_reset_aes        : std_logic;
    signal secret_in_buffer_out_valid         : std_logic;


    signal aes_data_out               : std_logic_vector(128-1 downto 0);
    signal aes_data_valid_ch0       : std_logic;
    signal aes_data_valid_ch1       : std_logic;
    signal AES_ch0_out_buffer_read : std_logic;
    signal share_in                   : share_in_bus_array;


    signal sharegen_in_buffer_header           : std_logic_vector(128 - 1 downto 0);
    signal sharegen_in_buffer_header_valid     : std_logic;
    signal sharegen_in_buffer_packet_available : std_logic;
    signal sharegen_in_buffer_secret           : std_logic_vector(M-1 downto 0);
    signal sharegen_in_buffer_secret_valid     : std_logic;


    signal sharegen_shares           : std_logic_vector((SHARE_NUM * M) - 1 downto 0);
    signal sharegen_shares_valid     : std_logic;
    signal ShareGen_secret_in_write : std_logic;
    signal sharegen_read_data        : std_logic;
    signal sharegen_mode             : std_logic;


    signal share_out_buffer_share_valid      : std_logic_vector(SHARE_NUM - 1 downto 0);
    signal share_out_buffer_share_last       : std_logic_vector(SHARE_NUM - 1 downto 0);
    signal share_out_buffer_packet_available : std_logic_vector(SHARE_NUM - 1 downto 0);
    signal share_out_buffer_share_out        : share_out_bus_array;


    signal share_in_buffer_share_last       : std_logic_vector(DEGREE downto 0);
    signal share_in_buffer_header_valid     : std_logic_vector(128-1 downto 0);
    signal share_in_buffer_header_out       : share_in_header_array;
    signal share_in_buffer_css_mode         : std_logic_vector(DEGREE downto 0);
    signal share_in_buffer_share_valid      : std_logic_vector(DEGREE downto 0);
    signal share_in_buffer_share_out        : share_in_buf_bus_array;
    signal share_in_buffer_packet_available : std_logic_vector(DEGREE downto 0);


    signal SecRec_secret_out               : std_logic_vector(M-1 downto 0);
    signal secrec_out_buffer_key_out       : std_logic_vector(128-1 downto 0);
    signal secrec_out_buffer_header_valid  : std_logic;
    signal secrec_out_buffer_header_out    : std_logic_vector(128-1 downto 0);
    signal secrec_css_mode                 : std_logic;
    signal secrec_share_in_port            : std_logic_vector(((DEGREE+1)*M)-1 downto 0);
    signal secrec_ready                    : std_logic;
    signal SecRec_secret_out_valid         : std_logic;
    signal SecRec_share_in_read            : std_logic;


    signal secrec_out_buffer_key_out_valid : std_logic;
    signal secrec_out_buffer_read          : std_logic;
    signal secrec_out_packet_ready           : std_logic;
    signal secrec_out_buffer_data_out      : std_logic_vector(128-1 downto 0);
    signal secrec_out_buffer_data_valid      : std_logic;
    signal secrec_out_buffer_last_word     : std_logic;


    signal secret_out_buffer_data_out         : std_logic_vector(EXTERNAL_BUS_WIDTH - 1 downto 0);
    signal secret_out_buffer_secret_out_valid : std_logic;
    signal secret_out_buffer_secret_out_last  : std_logic;
    signal secret_out_buffer_packet_available : std_logic;

    --nt header
    signal wr_nt_header          : std_logic_vector(SHARE_NUM - 1 + 1 downto 0);

    signal status_SecRec         : std_logic_vector((DEGREE + 1) * 2 + 2 - 1 downto 0);
    signal share_out_fifo_empty : std_logic_vector(SHARE_NUM - 1 downto 0);
    signal share_out_fifo_full  : std_logic_vector(SHARE_NUM - 1 downto 0);
    signal secret_in_fifo_empty : std_logic;
    signal secret_in_fifo_full  : std_logic;

    signal all_share_in_packets_ready    : std_logic;
    signal share                         : bus_array;

    signal share_in_packet_cnt           : share_in_bus_array;
    signal share_in_packet_cnt_rst       : std_logic;

begin

    ShareGen_Buff_rd_valid <= share_out_buffer_share_valid;
    ShareGen_Buff_last<= share_out_buffer_share_last;
    ShareGen_Buff_packet_ready <= share_out_buffer_packet_available;

    secret_out <= secret_out_buffer_data_out;
    secret_out_valid <= secret_out_buffer_secret_out_valid;
    secret_out_last <= secret_out_buffer_secret_out_last;
    secret_out_available <= secret_out_buffer_packet_available;
```

```
wr_nt_header <= sel_nt_header_i when wr_nt_header_i= '1' else (others => '0');

connect_share_out: for i in 0 to SHARE_NUM-1 generate
    share_out_port(((i+1)*EXTERNAL_BUS_WIDTH)-1 downto i*EXTERNAL_BUS_WIDTH)
                                                    <= share_out_buffer_share_out(i);
    share(i) <= sharegen_shares( ((i+1)*M)-1 downto i*M);
end generate;

connect_share_in: for i in 0 to DEGREE generate
    share_in(i) <= share_in_port( ((i+1)*EXTERNAL_BUS_WIDTH) -1 downto i*EXTERNAL_BUS_WIDTH);
    SecRec_share_in_port( ((i+1)*M) -1 downto i*M) <= share_in_buffer_share_out(i);
end generate;


secret_in_buffer_cmp : secret_in_buffer
    generic map(WIDTH_IN     => EXTERNAL_BUS_WIDTH,
                WIDTH_OUT    => 128,
                MAX_PACKETs => SECRET_IN_BUFFER_MAX_PACKETS,
                packet_size => (packet_size+header_size) )
    port map(clk                  => clk,
             rst                  => rst,
             data_in_i            => secret_in,
             data_in_valid_i      => secret_in_write,
             data_in_last_i       => secret_in_last,
             read_en_i            => secret_in_buffer_read,
             data_out_o           => secret_in_buffer_out_data,
             data_out_valid_o     => secret_in_buffer_out_valid,
             header_out_valid_o => secret_in_buffer_heade_out_valid,
             data_out_last_o      => secret_in_buffer_out_last,
             packet_available_o => secret_in_buffer_packet_available,
             reset_aes_o          => secret_in_buffer_reset_aes );

AES1 : AES
    generic map(SR_TYPE        => AES_SR_TYPE,
                COUNTER_WIDTH => AES_COUNTER_WIDTH)
    port map(clk                    => clk,
             rst                    => rst,
             plaintext_ch0_i        => secret_in_buffer_out_data,
             plaintext_ch1_i        => secrec_out_buffer_data_out,
             key_ch1_i              => secrec_out_buffer_key_out,
             new_key_i              => new_key,
             new_key_rd_o           => new_key_rd_o,
             Ciphertext_o           => aes_data_out,
             data_out_ch0_valid_o => aes_data_valid_ch0,
             data_in_ch0_valid_i  => secret_in_buffer_out_valid,
             data_out_ch1_valid_o => aes_data_valid_ch1,
             data_in_ch1_valid_i  => secrec_out_buffer_data_valid,
             reset_counter_ch0_i  => secret_in_buffer_reset_aes,
             reset_counter_ch1_i  => secrec_out_buffer_key_out_valid   );

--AES Verwaltung
AES_arbiter : AES_channel_arbiter
    port map (clk          => clk,
              rst          => rst,
              CH0_ready_i => secret_in_buffer_packet_available,
              CH0_last_i  => secret_in_buffer_out_last,
              Ch1_ready_i => secrec_out_packet_ready,
              CH1_last_i  => secrec_out_buffer_last_word,
              CH0_en_o     => secret_in_buffer_read,
              Ch1_en_o     => secrec_out_buffer_read );

share_gen_in_buffer : sharegen_in_buffer
    generic map(WIDTH_IN     => 128,
                WIDTH_OUT    => M,
                MAX_PACKETs => SHAREGEN_BUFFER_MAX_PACKETS,
                packet_size => packet_size,
                DEGREE          => DEGREE )
    port map(clk                     => clk,
             rst                     => rst,
             data_in_i               => aes_data_out,
             data_out_o              => sharegen_in_buffer_secret,
             rd_en_i                 => sharegen_read_data,
             data_in_valid_i         => AES_data_valid_ch0,
             data_out_valid_o        => sharegen_in_buffer_secret_valid,
             packet_available_o      => sharegen_in_buffer_packet_available,
             css_header_in_i         => secret_in_buffer_out_data,
             css_header_out_o        => sharegen_in_buffer_header,
             css_header_in_valid_i  => secret_in_buffer_heade_out_valid,
             css_header_out_valid_o => sharegen_in_buffer_header_valid,
             key_in_i                => new_key,
             sharegen_mode_o         => sharegen_mode );

share_gen : sharegen
    generic map(M          => M,
                x_BW       => x_BW,
                SHARE_NUM => SHARE_NUM,
                DEGREE     => DEGREE )
    port map(clk               => clk,
             rst               => rst,
             secret_i          => sharegen_in_buffer_secret,
             data_in_valid_i => sharegen_in_buffer_secret_valid,
             x_port_i          => x_port,
```

```
            shares_o         => sharegen_shares ,
            shares_valid_o   => sharegen_shares_valid ,
            rand_i           => rand ,
            rd_rand_o        => rand_rd_o ,
            mode_i           => sharegen_mode ,
            rd_data_o        => sharegen_read_data );


  Share_out_Buffer : for i in 0 to SHARE_NUM-1 generate
      b: out_buffer
          generic map(WIDTH_IN        => M,
                      WIDTH_OUT       => EXTERNAL_BUS_WIDTH,
                      MAX_PACKETs     => SHARE_OUT_BUFFER_MAX_PACKETS,
                      PACKET_SIZE     => (packet_size+128/8), --PACKET_SIZE+16 Bytes extra space for key
                      NT_HEADER_SIZE  => 11,
                      x_BW            => X_BW,
                      VERSION_NR      => 0,
                      BUFFER_TYPE     => "share" )
          port map(clk                  => clk ,
                   rst                  => rst ,
                   data_in_i            => share(i),
                   data_out_o           => share_out_buffer_share_out(i),
                   data_out_valid_o     => share_out_buffer_share_valid(i),
                   data_out_last_o      => share_out_buffer_share_last(i),
                   rd_data_i            => ShareGen_Buff_rd_en(i),
                   data_in_valid_i      => sharegen_shares_valid ,
                   packet_available_o   => share_out_buffer_packet_available(i),
                   css_header_i         => sharegen_in_buffer_header ,
                   css_header_valid_i   => sharegen_in_buffer_header_valid ,
                   x_value_i            => x_port((i+1)*X_BW - 1 downto i*X_BW),
                   nt_header_i          => nt_header_i ,
                   nt_header_valid_i    => wr_nt_header(i),
                   rst_nt_header_cnt_i  => rst_nt_header_cnt_i );
  end generate;

  share_in_buffer_cmp : for i in 0 to DEGREE generate
      b: share_in_buffer
      generic map(WIDTH_IN     => EXTERNAL_BUS_WIDTH,
                  WIDTH_OUT    => M,
                  MAX_PACKETs  => SHARE_IN_BUFFER_MAX_PACKETS,
                  PACKET_SIZE  => PACKET_SIZE )
      port map(clk                 => clk ,
               rst                 => rst ,
               data_in_i           => share_in(i),
               data_in_valid_i     => share_in_write(i),
               rd_data_i           => SecRec_share_in_read ,
               data_out_o          => share_in_buffer_share_out(i),
               data_out_valid_o    => share_in_buffer_share_valid(i),
               data_out_last_o     => share_in_buffer_share_last(i),
               header_out_o        => share_in_buffer_header_out(i),
               header_out_valid_o  => share_in_buffer_header_valid(i),
               packet_available_o  => share_in_buffer_packet_available(i),
               css_mode_o          => share_in_buffer_css_mode(i),
               packet_cnt_o        => share_in_packet_cnt(i),
               rst_packet_cnt_i    => share_in_packet_cnt_rst );
  end generate;

  secrec : secret_reconstruction
      generic map(M                => M,
                  SHARES           => DEGREE+1,
                  DSPperKaratsuba  => DSPPERKARATSUBA,
                  USEDSP           => USEDSP,
                  PIPELINING       => PIPELINING,
                  MATRIX_MODE      => MATRIX_MODE )
      port map(clk                 => clk ,
               rst                 => rst ,
               matrix_port_i       => matrix_port ,
               matrix_coef_i       => matrix_coef_i ,
               matrix_coef_valid_i => matrix_coef_valid_i ,
               coef_sel_share_i    => coef_sel_share_i ,
               coef_sel_pos_i      => coef_sel_pos_i ,
               share_port_i        => SecRec_share_in_port ,
               secret_o            => secrec_secret_out ,
               secret_valid_o      => secrec_secret_out_valid ,
               share_valid_i       => share_in_buffer_share_valid ,
               css_mode_i          => share_in_buffer_css_mode(0),
               css_mode_o            => secrec_css_mode ,
               ready_o             => secrec_ready     );

  sec_rec_out_buffer : secrec_out_buffer
      generic map(WIDTH_IN     => M,
                  WIDTH_OUT    => 128,
                  MAX_PACKETs  => SECREC_BUFFER_MAX_PACKETS,
                  packet_size  => packet_size ,
                  DEGREE       => DEGREE )
      port map(clk                    => clk ,
               rst                    => rst ,
               data_in_i              => secrec_secret_out ,
               data_out_o             => secrec_out_buffer_data_out ,
               read_en_i              => secrec_out_buffer_read ,
               data_in_valid_i        => secrec_secret_out_valid ,
               data_out_valid_o       => secrec_out_buffer_data_valid ,
```

```
                packet_available_o      => secrec_out_packet_ready,
                css_header_in_i         => share_in_buffer_header_out(0),
                css_header_out_o        => secrec_out_buffer_header_out,
                css_header_in_valid_i   => share_in_buffer_header_valid(0),
                css_header_out_valid_o  => secrec_out_buffer_header_valid,
                key_out_o               => secrec_out_buffer_key_out,
                data_out_last_o         => secrec_out_buffer_last_word,
                key_out_valid_o            => secrec_out_buffer_key_out_valid,
                input_is_keyn_i            => secrec_css_mode );

    Secret_out_Buffer : out_buffer
        generic map(WIDTH_IN        => 128,
                    WIDTH_OUT       => EXTERNAL_BUS_WIDTH,
                    MAX_PACKETs     => SECRET_OUT_BUFFER_MAX_PACKETS,
                    PACKET_SIZE     => packet_size,
                    NT_HEADER_SIZE  => 11,
                    X_BW            => X_BW,
                    VERSION_NR      => 0,
                    BUFFER_TYPE     => "secret" )
        port map(clk                 => clk,
                 rst                 => rst,
                 data_in_i           => aes_data_out,
                 data_out_o          => secret_out_buffer_data_out,
                 data_out_valid_o    => secret_out_buffer_secret_out_valid,
                 data_out_last_o        => secret_out_buffer_secret_out_last,
                 rd_data_i           => secret_out_read,
                 data_in_valid_i     => AES_data_valid_ch1,
                 packet_available_o  => secret_out_buffer_packet_available,
                 css_header_i        => secrec_out_buffer_header_out,
                 css_header_valid_i  => secrec_out_buffer_header_valid,
                 x_value_i           => (others => '0'),
                 nt_header_i         => nt_header_i,
                 nt_header_valid_i   => wr_nt_header(SHARE_NUM),
                 rst_nt_header_cnt_i => rst_nt_header_cnt_i );

    process(share_in_buffer_packet_available) is
    begin
        all_share_in_packets_ready <=  '1';
        for i in 0 to DEGREE loop
            if(share_in_buffer_packet_available(i) = '0') then
                all_share_in_packets_ready <=  '0';
            end if;
        end loop;
    end process;

    secrec_share_in_read <= secrec_ready and all_share_in_packets_ready;

    ShareGen_secret_in_write <= sharegen_in_buffer_secret_valid;
    AES_ch0_out_buffer_read <= '1';

    process(CSS_Ctrl(8-1 downto 0)) is
        variable temp_out : std_logic_vector(32-1 downto 0);
    begin
        temp_out(status_SecRec'length-1 downto 0) := status_SecRec;
        temp_out(32-1 downto status_SecRec'length) := (others => '0');
        for i in 0 to DEGREE loop
            if  to_integer(unsigned(CSS_Ctrl(8-1 downto 0))) = i then
                temp_out := share_in_packet_cnt(i);
            end if;
        end loop;

        SecRec_Status <= temp_out;

        if CSS_Ctrl(8-1 downto 0) = x"FF" then
            share_in_packet_cnt_rst <= '1';
        else
            share_in_packet_cnt_rst <= '0';
        end if;
    end process;


    ShareGen_Status(SHARE_NUM * 2 + 2 - 1 downto 0)  <= share_out_fifo_empty & share_out_fifo_full
                                                       & secret_in_fifo_empty & secret_in_fifo_full;
    ShareGen_Status(31 downto SHARE_NUM * 2 + 2)     <= (others => '0');


end Behavioral;
```

8×8 bit polynomial multiplier, using 1 DSP

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity DSPMult8Bit is
    GENERIC (Pipelining : integer := 1);
    PORT(clk : in std_logic;
         ain : in std_logic_vector(8-1 downto 0);
         bin : in std_logic_vector(8-1 downto 0);
         d   : out std_logic_vector(15-1 downto 0) );
end DSPMult8Bit;

architecture Behavioral of DSPMult8Bit is

    signal a         : std_logic_vector(8-1 downto 0);
    signal b         : std_logic_vector(8-1 downto 0);
    signal DSPina    : std_logic_vector(25-1 downto 0) := (others => '0');
    signal DSPinb    : std_logic_vector(16-1 downto 0):= (others => '0');
    signal DSPout    : std_logic_vector(41-1 downto 0):= (others => '0');
    signal DSPoutPol : std_logic_vector(13-1 downto 0):= (others => '0');

    function VECTORAND(a : std_logic_vector(8-1 downto 0); b: std_logic)
        return std_logic_vector is
    begin
        if(b='1') then
            return a;
        else
            return "00000000";
        end if;
    end VECTORAND;

begin

    --make connections to use normal Mult with DSP
    d <= "00" & DSPoutPol xor
          "0" & VECTORAND(a, b(6)) & "000000" xor
                VECTORAND(a, b(7)) & "0000000";

    process(ain, bin) is
    begin
        for i in 0 to 8-1 loop
            DSPina(i*3) <= ain(i);
        end loop;
        for i in 0 to 6-1 loop
            DSPinb(i*3) <= bin(i);
        end loop;
    end process;

    p1: if(Pipelining > 0) generate
        process(clk) is
        begin
            if(clk'event and clk='1') then
                for i in 0 to (8+6-1)-1 loop
                    DSPoutPol(i) <= DSPout(i*3);
                end loop;
                a <= ain;
                b <= bin;
            end if;
        end process;
    end generate;

    p2: if(Pipelining = 0) generate
        process(DSPout) is
        begin
            for i in 0 to (8+6-1)-1 loop
                DSPoutPol(i) <= DSPout(i*3);
            end loop;
        end process;
        a <= ain;
        b <= bin;
    end generate;

    DSPout <= std_logic_vector(unsigned(DSPina) * unsigned(DSPinb));

end Behavioral;
```

16×16 bit polynomial multiplier, using 2 DSPs

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity DSPMult16Bitv2 is
    GENERIC(Pipelining : integer := 0);
    PORT(clk     : in std_logic;
         ain     : in std_logic_vector(16-1 downto 0);
         bin     : in std_logic_vector(16-1 downto 0);
         result  : out std_logic_vector(31-1 downto 0) );
end DSPMult16Bitv2;

architecture Behavioral of DSPMult16Bitv2 is
    signal DSP1ina: std_logic_vector(25-1 downto 0) := (others => '0');
    signal DSP1inb: std_logic_vector(16-1 downto 0):= (others => '0');
    signal DSP1out: std_logic_vector(41-1 downto 0):= (others => '0');
    signal DSP1outPol : std_logic_vector(14-1 downto 0):= (others => '0');

    signal DSP2ina: std_logic_vector(25-1 downto 0) := (others => '0');
    signal DSP2inb: std_logic_vector(16-1 downto 0):= (others => '0');
    signal DSP2out: std_logic_vector(41-1 downto 0):= (others => '0');
    signal DSP2outPol : std_logic_vector(14-1 downto 0):= (others => '0');

    signal a : std_logic_vector(16-1 downto 0);
    signal b : std_logic_vector(16-1 downto 0);
    signal c : std_logic_vector(31-1 downto 0);

    function VECTORAND7(a : std_logic_vector(7-1 downto 0); b: std_logic)
                  return std_logic_vector is
    begin
      if(b='1') then
        return a;
      else
        return "0000000";
     end if;
    end VECTORAND7;

    function VECTORAND6(a : std_logic_vector(6-1 downto 0); b: std_logic)
                      return std_logic_vector is
        begin
          if(b='1') then
            return a;
          else
            return "000000";
         end if;
        end VECTORAND6;

    function VECTORAND16(a : std_logic_vector(16-1 downto 0); b: std_logic)
                      return std_logic_vector is
        begin
          if(b='1') then
            return a;
          else
            return "0000000000000000";
         end if;
        end VECTORAND16;

begin

    --make connections to use normal Mult with DSP
    process(ain, bin) is
    begin
        for i in 0 to 9-1 loop
            DSP1ina(i*3) <= ain(i+1);
            DSP2ina(i*3) <= ain(i+6);
        end loop;
        for i in 0 to 6-1 loop
            DSP1inb(i*3) <= bin(i);
            DSP2inb(i*3) <= bin(i+10);
        end loop;
    end process;

    p1: if (Pipelining > 0) generate
        process(clk) is
        begin
            if(clk'event and clk='1') then
                for i in 0 to (9+6-1)-1 loop
                    DSP1outPol(i) <= DSP1out(i*3);
                    DSP2outPol(i) <= DSP2out(i*3);
                end loop;
                a <= ain;  --one pipeline stages for the others
                b <= bin;  --values as well
            end if;
        end process;
    end generate;

    p2: if(Pipelining = 0) generate
        process(DSP1out, DSP2out) is
        begin
            for i in 0 to (9+6-1)-1 loop
```

```
                DSP1outPol(i) <= DSP1out(i*3);
                DSP2outPol(i) <= DSP2out(i*3);
            end loop;
        end process;
        a <= ain;  --straight through, without pipeline
        b <= bin;
    end generate;

    DSP1out <= std_logic_vector(unsigned(DSP1ina) * unsigned(DSP1inb));
    DSP2out <= std_logic_vector(unsigned(DSP2ina) * unsigned(DSP2inb));

    c <=
   "000000000000000" & VECTORAND6(a(16-1 downto 10), b(0)) & "000000000" & (a(0) and b(0)) xor
    "00000000000000" & VECTORAND6(a(16-1 downto 10), b(1)) & "000000000" & (a(0) and b(1)) & "0" xor
     "0000000000000" & VECTORAND6(a(16-1 downto 10), b(2)) & "000000000" & (a(0) and b(2)) & "00" xor
      "000000000000" & VECTORAND6(a(16-1 downto 10), b(3)) & "000000000" & (a(0) and b(3)) & "000" xor
       "00000000000" & VECTORAND6(a(16-1 downto 10), b(4)) & "000000000" & (a(0) and b(4)) & "0000" xor
        "0000000000" & VECTORAND6(a(16-1 downto 10), b(5)) & "000000000" & (a(0) and b(5)) & "00000" xor
         "000000000" & VECTORAND16(a(16-1 downto 0), b(6)) & "000000" xor
          "00000000" & VECTORAND16(a(16-1 downto 0), b(7)) & "0000000" xor
           "0000000" & VECTORAND16(a(16-1 downto 0), b(8)) & "00000000" xor
            "000000" & VECTORAND16(a(16-1 downto 0), b(9)) & "000000000" xor

   "00000" & (a(15) and b(10)) & "000000000" & VECTORAND6(a(6-1 downto 0), b(10)) & "0000000000" xor
    "0000" & (a(15) and b(11)) & "000000000" & VECTORAND6(a(6-1 downto 0), b(11)) & "00000000000" xor
     "000" & (a(15) and b(12)) & "000000000" & VECTORAND6(a(6-1 downto 0), b(12)) & "000000000000" xor
      "00" & (a(15) and b(13)) & "000000000" & VECTORAND6(a(6-1 downto 0), b(13)) & "0000000000000" xor
       "0" & (a(15) and b(14)) & "000000000" & VECTORAND6(a(6-1 downto 0), b(14)) & "00000000000000" xor
             (a(15) and b(15)) & "000000000" & VECTORAND6(a(6-1 downto 0), b(15)) & "000000000000000" xor
                          "000000000000000" &  DSP1outPol                         & "0"             xor
                                        "0" &  DSP2outPol                         & "000000000000000";

   result <= c;

end Behavioral;
```

16×16 bit polynomial multiplier, using 3 DSPs

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity DSPMult16Bitv3 is
    GENERIC (Pipelining : integer := 1);
    PORT(clk     : in std_logic;
         ain     : in std_logic_vector(16-1 downto 0);
         bin     : in std_logic_vector(16-1 downto 0);
         result  : out std_logic_vector(31-1 downto 0) );
end DSPMult16Bitv3;

architecture Behavioral of DSPMult16Bitv3 is
    signal c : std_logic_vector(31-1 downto 0);
    signal a : std_logic_vector(16-1 downto 0);
    signal b : std_logic_vector(16-1 downto 0);

    signal DSP1ina    : std_logic_vector(25-1 downto 0) := (others => '0');
    signal DSP1inb    : std_logic_vector(16-1 downto 0):= (others => '0');
    signal DSP1out    : std_logic_vector(41-1 downto 0):= (others => '0');
    signal DSP1outPol : std_logic_vector(14-1 downto 0):= (others => '0');

    signal DSP2ina    : std_logic_vector(25-1 downto 0) := (others => '0');
    signal DSP2inb    : std_logic_vector(16-1 downto 0):= (others => '0');
    signal DSP2out    : std_logic_vector(41-1 downto 0):= (others => '0');
    signal DSP2outPol : std_logic_vector(14-1 downto 0):= (others => '0');

    signal DSP3ina    : std_logic_vector(25-1 downto 0) := (others => '0');
    signal DSP3inb    : std_logic_vector(16-1 downto 0):= (others => '0');
    signal DSP3out    : std_logic_vector(41-1 downto 0):= (others => '0');
    signal DSP3outPol : std_logic_vector(14-1 downto 0):= (others => '0');


    function VECTORAND7(a : std_logic_vector(7-1 downto 0); b: std_logic)
        return std_logic_vector is
    begin
        if(b='1') then
            return a;
        else
            return "0000000";
        end if;
    end VECTORAND7;

    function VECTORAND9(a : std_logic_vector(9-1 downto 0); b: std_logic)
        return std_logic_vector is
    begin
        if(b='1') then
            return a;
        else
            return "000000000";
        end if;
    end VECTORAND9;

    function VECTORAND16(a : std_logic_vector(16-1 downto 0); b: std_logic)
        return std_logic_vector is
    begin
        if(b='1') then
            return a;
        else
            return "0000000000000000";
        end if;
    end VECTORAND16;

begin

    --make connections to use normal Mult with DSP
    process(ain, bin) is
    begin
        for i in 0 to 9-1 loop
            DSP1ina(i*3) <= ain(i);
            DSP2ina(i*3) <= ain(i+7);
            DSP3ina(i*3) <= bin(i+6);
        end loop;
        for i in 0 to 6-1 loop
            DSP1inb(i*3) <= bin(i);
            DSP2inb(i*3) <= bin(i+10);
            DSP3inb(i*3) <= ain(i+1);
        end loop;
    end process;

    p1: if (Pipelining > 0) generate
        process(clk) is
        begin
            if(clk'event and clk='1') then
                for i in 0 to (9+6-1)-1 loop
                    DSP1outPol(i) <= DSP1out(i*3);
                    DSP2outPol(i) <= DSP2out(i*3);
                    DSP3outPol(i) <= DSP3out(i*3);
                end loop;
                a <= ain;  --one pipeline stages for the others
```

```
                b <= bin;  --values as well
            end if;
        end process;
    end generate;

    p2: if(Pipelining = 0) generate
        process(DSP1out, DSP2out, DSP3out) is
        begin
            for i in 0 to (9+6-1)-1 loop
                DSP1outPol(i) <= DSP1out(i*3);
                DSP2outPol(i) <= DSP2out(i*3);
                DSP3outPol(i) <= DSP3out(i*3);
            end loop;
        end process;
        a <= ain;  --straight through, without pipeline
        b <= bin;
    end generate;

    DSP1out <= std_logic_vector(unsigned(DSP1ina) * unsigned(DSP1inb));
    DSP2out <= std_logic_vector(unsigned(DSP2ina) * unsigned(DSP2inb));
    DSP3out <= std_logic_vector(unsigned(DSP3ina) * unsigned(DSP3inb));

  c <=
  "000000000000000" & VECTORAND7(a(16-1 downto 9), b(0))  & "000000000" xor
   "00000000000000" & VECTORAND7(a(16-1 downto 9), b(1))  & "0000000000" xor
    "0000000000000" & VECTORAND7(a(16-1 downto 9), b(2))  & "00000000000" xor
     "000000000000" & VECTORAND7(a(16-1 downto 9), b(3))  & "000000000000" xor
      "00000000000" & VECTORAND7(a(16-1 downto 9), b(4))  & "0000000000000" xor
       "0000000000" & VECTORAND7(a(16-1 downto 9), b(5))  & "00000000000000" xor

        "000000000" & VECTORAND9(a(16-1 downto 7), b(6)) & "000000" & (a(0) and b(6)) & "000000" xor
         "00000000" & VECTORAND9(a(16-1 downto 7), b(7)) & "000000" & (a(0) and b(7)) & "0000000" xor
          "0000000" & VECTORAND9(a(16-1 downto 7), b(8)) & "000000" & (a(0) and b(8)) & "00000000" xor
           "000000" & VECTORAND9(a(16-1 downto 7), b(9)) & "000000" & (a(0) and b(9)) & "000000000" xor

      "00000000000000000000" & (a(0) and b(10)) & "0000000000"  xor
       "0000000000000000000" & (a(0) and b(11)) & "00000000000"  xor
        "000000000000000000" & (a(0) and b(12)) & "000000000000"  xor
         "00000000000000000" & (a(0) and b(13)) & "0000000000000"  xor
          "0000000000000000" & (a(0) and b(14)) & "00000000000000"  xor
           "000000000" &  VECTORAND7(a(7-1 downto 0), b(15))  & "000000000000000" xor
       "00000000000000000" &  DSP1outPol                                            xor
                              DSP2outPol & "00000000000000000"                     xor
              "0000000000" &  DSP3outPol & "0000000" ;

        result <= c;

end Behavioral;
```

16 × 16 bit polynomial multiplier, using 4 DSPs

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity DSPMult16Bitv4 is
    generic(Pipelining : integer := 0);
    port(clk     : in std_logic;
         ain     : in std_logic_vector(16-1 downto 0);
         bin     : in std_logic_vector(16-1 downto 0);
         result  : out std_logic_vector(31-1 downto 0)

         );
end DSPMult16Bitv4;

architecture Behavioral of DSPMult16Bitv4 is
    signal a : std_logic_vector(16-1 downto 0);
    signal b : std_logic_vector(16-1 downto 0);
    signal c : std_logic_vector(31-1 downto 0);

    signal DSP1ina    : std_logic_vector(25-1 downto 0) := (others => '0');
    signal DSP1inb    : std_logic_vector(16-1 downto 0) := (others => '0');
    signal DSP1out    : std_logic_vector(41-1 downto 0) := (others => '0');
    signal DSP1oP : std_logic_vector(14-1 downto 0) := (others => '0');

    signal DSP2ina    : std_logic_vector(25-1 downto 0) := (others => '0');
    signal DSP2inb    : std_logic_vector(16-1 downto 0) := (others => '0');
    signal DSP2out    : std_logic_vector(41-1 downto 0) := (others => '0');
    signal DSP2oP : std_logic_vector(14-1 downto 0) := (others => '0');

    signal DSP3ina    : std_logic_vector(25-1 downto 0) := (others => '0');
    signal DSP3inb    : std_logic_vector(16-1 downto 0) := (others => '0');
    signal DSP3out    : std_logic_vector(41-1 downto 0) := (others => '0');
    signal DSP3oP : std_logic_vector(14-1 downto 0) := (others => '0');

    signal DSP4ina    : std_logic_vector(25-1 downto 0) := (others => '0');
    signal DSP4inb    : std_logic_vector(16-1 downto 0) := (others => '0');
    signal DSP4out    : std_logic_vector(41-1 downto 0) := (others => '0');
    signal DSP4oP : std_logic_vector(14-1 downto 0) := (others => '0');

begin

    --make connections to use normal Mult with DSP
    process(ain, bin) is
    begin
        for i in 0 to 9-1 loop
            DSP1ina(i*3) <= ain(i);
            DSP2ina(i*3) <= ain(i+7);
            DSP3ina(i*3) <= bin(i+6);
            DSP4ina(i*3) <= bin(i+1);
        end loop;
        for i in 0 to 6-1 loop
            DSP1inb(i*3) <= bin(i);
            DSP2inb(i*3) <= bin(i+10);
            DSP3inb(i*3) <= ain(i+1);
            DSP4inb(i*3) <= ain(i+9);
        end loop;
    end process;

    p1: if (Pipelining > 0) generate
        process(clk) is
        begin
            if(clk'event and clk='1') then
                for i in 0 to (9+6-1)-1 loop
                    DSP1oP(i) <= DSP1out(i*3);
                    DSP2oP(i) <= DSP2out(i*3);
                    DSP3oP(i) <= DSP3out(i*3);
                    DSP4oP(i) <= DSP4out(i*3);
                end loop;
                a <= ain;  --one pipeline stages for the others
                b <= bin;  --values as well
            end if;
        end process;
    end generate;

    p2: if(Pipelining = 0) generate
        process(DSP1out, DSP2out, DSP3out, DSP4out) is
        begin
            for i in 0 to (9+6-1)-1 loop
                DSP1oP(i) <= DSP1out(i*3);
                DSP2oP(i) <= DSP2out(i*3);
                DSP3oP(i) <= DSP3out(i*3);
                DSP4oP(i) <= DSP4out(i*3);
            end loop;
        end process;
        a <= ain;  --straight through, without pipeline
        b <= bin;
    end generate;

    DSP1out <= std_logic_vector(unsigned(DSP1ina) * unsigned(DSP1inb));
    DSP2out <= std_logic_vector(unsigned(DSP2ina) * unsigned(DSP2inb));
```

```
    DSP3out <= std_logic_vector(unsigned(DSP3ina) * unsigned(DSP3inb));
    DSP4out <= std_logic_vector(unsigned(DSP4ina) * unsigned(DSP4inb));


    c(5 downto 0) <= DSP1oP(5 downto 0);

    c(6)  <= DSP1oP(6) XOR (a(0) AND b(6));
    c(7)  <= DSP1oP(7) XOR DSP3oP(0) XOR (a(0) AND b(7));
    c(8)  <= DSP1oP(8) XOR DSP3oP(1) XOR (a(0) AND b(8));
    c(9)  <= (a(9)  AND b(0)) XOR DSP1oP(9) XOR DSP3oP(2) XOR (a(0) AND b(9));
    c(10) <= (a(10) AND b(0)) XOR DSP4oP(0) XOR DSP1oP(10) XOR DSP3oP(3) XOR (a(0) AND b(10));
    c(11) <= (a(11) AND b(0)) XOR DSP4oP(1) XOR DSP1oP(11) XOR DSP3oP(4) XOR (a(0) AND b(11));
    c(12) <= (a(12) AND b(0)) XOR DSP4oP(2) XOR DSP1oP(12) XOR DSP3oP(5) XOR (a(0) AND b(12));
    c(13) <= (a(13) AND b(0)) XOR DSP4oP(3) XOR DSP1oP(13) XOR (a(7) AND b(6)) XOR DSP3oP(6)
             XOR ( a(0) AND b(13));
    c(14) <= (a(14) AND b(0)) XOR DSP4oP(4) XOR (a(8) AND b(6)) XOR (a(7) AND b(7)) XOR DSP3oP(7)
             XOR (a(0) AND b(14));
    c(15) <= (a(15) AND b(0)) XOR DSP4oP(5) XOR (a(8) AND b(7)) XOR (a(7) AND b(8)) XOR DSP3oP(8)
             XOR (a(0) AND b(15));
    c(16) <= (a(15) AND b(1)) XOR DSP4oP(6) XOR (a(8) AND b(8)) XOR (a(7) AND b(9)) XOR DSP3oP(9)
             XOR (a(1) AND b(15));
    c(17) <= (a(15) AND b(2)) XOR DSP4oP(7) XOR (a(8) AND b(9)) XOR DSP2oP(0) XOR DSP3oP(10)
             XOR ( a(2) AND b(15) );
    c(18) <= (a(15) AND b(3)) XOR DSP4oP(8) XOR DSP2oP(1) XOR DSP3oP(11) XOR (a(3) AND b(15));
    c(19) <= (a(15) AND b(4)) XOR DSP4oP(9) XOR DSP2oP(2) XOR DSP3oP(12) XOR (a(4) AND b(15));
    c(20) <= (a(15) AND b(5)) XOR DSP4oP(10) XOR DSP2oP(3) XOR DSP3oP(13) XOR (a(5) AND b(15));
    c(21) <= (a(15) AND b(6)) XOR DSP4oP(11) XOR DSP2oP(4) XOR (a(6) AND b(15) );
    c(22) <= (a(15) AND b(7)) XOR DSP4oP(12) XOR DSP2oP(5);
    c(23) <= (a(15) AND b(8)) XOR DSP4oP(13) XOR DSP2oP(6);
    c(24) <= (a(15) AND b(9)) XOR DSP2oP(7);

    c(30 downto 25) <= DSP2oP(13 downto 8);

    result <= c;

end Behavioral;
```

TRNG

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity TRNG_RO is
    Generic(number_of_RO   : integer := 11;
            number_of_INV  : integer :=3;
            use_constant   : boolean := false);
    Port ( clk       : in std_logic;
           rand_o    : out std_logic);
end TRNG_RO;

architecture Behavioral of TRNG_RO is

attribute dont_touch : string;

signal ro_slv: std_logic_vector(number_of_RO-1 downto 0);
signal ro_buf_slv: std_logic_vector(number_of_RO-1 downto 0);

attribute dont_touch of ro_slv : signal is "true";
attribute dont_touch of ro_buf_slv : signal is "true";

component RO is
    Generic(depth   : integer := 3);
    Port ( rand_o : out STD_LOGIC);
end component;

begin

    g: for i in 0 to number_of_RO-1 generate
        ro:  RO generic map(depth  => number_of_INV)
                port map (rand_o => ro1_slv(i));
    end generate;

    process(clk) is
        variable rand : std_logic := '0';
    begin
        if clk'event and clk='1' then
            ro_buf_slv <= ro_slv;
            rand := '0';
            for i in 0 to number_of_RO-1 loop
                rand := rand XOR ro_buf_slv(i);
            end loop;
           if(use_constant = true) then
                 rand_o <= '1';
           else
                rand_o <= rand;
           end if;
       end if;
    end process;

end Behavioral;
------------------------- Ring Oszillator ---------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RO is
    Generic(depth   : integer := 3);
    Port ( rand_o : out STD_LOGIC);
end RO;

architecture Behavioral of RO is

attribute dont_touch : string;
signal connect  : std_logic_vector(depth-1 downto 0);
attribute dont_touch of connect : signal is "true";

begin
    rand_o <= connect(0);
    connect(depth-1) <= not connect(0);
    g: for i in 0 to depth-2 generate
        connect(i) <= not connect(i+1);
    end generate;
end Behavioral;
```

D - C Code snapshots

Matrix inversion on the PS

This code is performed on the PS to first generate the matrix with the applied x-values and then it's inversion for the secret reconstruction. The algorithm described in [30] is applied with binary extension field operations of the library [26].

```
void ff_invMatrix(ff_element A[K][K]){

  ff_element A1[K][K];
  ff_element ff_mul_temp;
  //Step 1
  int p, i, j;
  //Step 2
  for(p=0; p<K; p++){
    //Step 3
    if(ff_eq(A[p][p], ff_zero)){
      printf("inversion not possible...\n");
        break;
    }
    //Step 5
    for(j=0; j<K; j++){
      if(j != p){
        ff_div(A[p][j], A[p][p], A1[p][j]);
      }
    }
    //Step 6
    for(i=0; i<K; i++){
      if(i != p){
        ff_div(A[i][p], A[p][p], A1[i][p]);
      }
    }
    //Step 7
    for(i=0; i<K; i++){
      for(j=0; j<K; j++){
        if(i != p && j != p){
          ff_mul(A[p][j], A1[i][p], ff_mul_temp);
          ff_add(A[i][j], ff_mul_temp, A1[i][j]);
        }
      }
    }
    //Step 8
    ff_inv(A[p][p], A1[p][p]);

    //copy new matrix
    for(i=0; i<K; i++){
      for(j=0; j<K; j++){
        ff_copy(A1[i][j], A[i][j]);
      }
    }
  }
}
```

Theoretical maximal throughput estimation for the share generation on the PC

This code only performs the required finite field calculations for the share generation, using the library [26]. The required random numbers can be either static or generated with the *rand*() function.

```
void CSS_ShareGenSpeedTest(int bit_width, int nr_shares){
    int i, j, cnt, n, sub_count;

    #if (GF == 16 || GF == 8)
        int secret[K];
        int share[nr_shares];
        int x_n[nr_shares];
        //initialisieren
        for(i=0; i<K; i++){
            secret[i] = 0;
        }
        #if USERAND >= 1
            for(i=0;i<nr_shares;i++){
                share[i] = 0x56;
                x_n[i] = 0x56;
            }
        #else
            for(i=0;i<nr_shares;i++){
                share[i] = 0x56;
                x_n[i] = 0x56;
            }
        #endif
    #else
        ff_element ff_share[10];
        ff_element ff_x_n[10];
        ff_element ff_secret_in[4];
        ff_element ff_secret_out[K];
        //initialisieren
        #if USERAND >= 1
            for(i=0; i<K; i++){
                ff_secret_in[i]->data[0]=rand();
                for(j=1;j<BLOCKS;j++){
                    ff_secret_in[i]->data[j]=rand();
                }
            }
            for(i=0;i<nr_shares;i++){
                ff_x_n[i]->data[0]=rand();
                for(j=1;j<BLOCKS;j++){
                    ff_x_n[i]->data[j]=0;
                }
            }
        #endif
    #endif


    #if GF >= 32
        for(i=0;i<nr_shares;i++){
            ff_copy(ff_zero, ff_share[i]);
        }
    #endif

    for(i=0;i<K;i++){
        //AES
        SimulateAES();

        //Share generation
        //iteration to reach 128 bits proccessd(according to AES output)
        for(sub_count = 0; sub_count<(128/bit_width); sub_count++){
           for(j=0; j<nr_shares;j++){
                #if GF >=32
                    ff_mul(ff_share[j], ff_x_n[j], ff_share[j]); //share=share*x
                    ff_add(ff_share[j], ff_secret_in[i], ff_share[j]); //share=share+secret
                #elif GF == 16
                    share[j]=galois_logtable_multiply(share[j], x_n[j], 16)^secret[i];
                #elif GF == 8
                    share[j]=galois_multtable_multiply(share[j], x_n[j], 8)^secret[i];
                #endif
           }
        }
    }
}
```

Theoretical maximal throughput estimation for the secret reconstruction on the PC

This code only performs the required finite field calculations for the secret reconstruction, using the library [26]. The applied x-values can be either static or generated with the $rand()$ function.

```
void CSS_SecRecSpeedTest(int bit_width, int nr_shares){
    int i, j, cnt, n, sub_count;
    //Initialisieren
    #if (GF == 16 || GF == 8)
        int secret[K];
        int share[nr_shares];
        int x_n[nr_shares];
        int inv_matrix[K][K];
        int result;
        //initialisieren
        #if USERAND >= 1
            for(i=0;i<grad;i++){
                for(j=0;j<K;j++){
                    inv_matrix[i][j]=rand();
                }
                share[i]=rand();
            }
            for(i=0;i<nr_shares;i++){
                share[i] = rand();
                x_n[i] = rand();
            }
        #else
            for(i=0;i<K;i++){
                for(j=0;j<K;j++){
                    inv_matrix[i][j]=89;//rand()+15;
                }
                share[i]=16;
            }
            for(i=0;i<nr_shares;i++){
                share[i] = 0x56;
                x_n[i] = 0x56;
            }
        #endif
    #else
        ff_element ff_A[K][K];
        ff_element ff_mul_temp;
        ff_element ff_share[K];
        ff_element ff_x_n[K];
        ff_element ff_secret_out[K];
        //initialisieren
        #if USERAND == 0
            for(i=0;i<nr_shares;i++){
                ff_x_n[i]->data[0]=0x48;
                for(j=1;j<BLOCKS;j++){
                    ff_x_n[i]->data[j]=0;
                }
            }
        #else
            for(i=0;i<nr_shares;i++){
                ff_x_n[i]->data[0]=rand();
                for(j=1;j<BLOCKS;j++){
                    ff_x_n[i]->data[j]=0;
                }
            }
        #endif
    #endif

    //Secrets Reconstruction
    for(j=0; j<K; j++){
         //diunds to precess 128 bit in total, for aes
         for(sub_count = 0; sub_count<(128/bit_width); sub_count++){
            #if GF >= 32
                ff_copy(ff_zero, ff_secret_out[j]);
            #else
                secret[j]=0;
            #endif
            for(i=0;i<K;i++){
                #if GF == 16
                    secret[j] = galois_logtable_multiply(inv_matrix[j][i], share[i], 16)^secret[j];
                #elif GF == 8
                    secret[j] = galois_multtable_multiply(inv_matrix[j][i], share[i], 8)^secret[j];
                #elif GF >= 32
                    ff_mul(ff_A[j][i], ff_share[i], ff_mul_temp);
                    ff_add(ff_secret_out[j], ff_mul_temp, ff_secret_out[j]);
                #endif
            }
        }
        //calc AES
        SimulateAES();
    }

}
```

Literature

[1] The Ethernet: A Local Area Network: Data Link Layer and Physical Layer Specifications . Specification 3, July 1981.

[2] Announcing the ADVANCED ENCRYPTION STANDARD (AES) . Federal Information, Processing Standards Publication 197, National Institute of Standards and Technology, February 2001.

[3] SECURE HASH STANDARD. Federal Information Processing Standards Publication 180-2, National Institute of Standards and Technology, 2002.

[4] Common RAID Disk Data Format Specification. SINA Specification, Storage Networking Industry Association, 2006.

[5] Modes of operation for an n-bit block cipher . ISO/IEC Standard 10116:2006, International Organization for Standardization, 2006.

[6] SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions . Federal Information Processing Standards Publication 202, National Institute of Standards and Technology, 2015.

[7] A. Abdallah and M. Salleh. Secret sharing scheme security and performance analysis. In *2015 International Conference on Computing, Control, Networking, Electronics and Embedded Systems Engineering (ICCNEEE)*, pages 173–180, Sept 2015.

[8] Amazon Web Services - Whitepaper. Accessed: 01.04. 2017. `https://d0.awsstatic.com/whitepapers/Security/AWS_Security_Best_Practices.pdf/`.

[9] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 598–609, New York, NY, USA, 2007. ACM.

[10] George R. Blakley. Safeguarding Cryptographic Keys. In *Proceedings of the 1979 AFIPS National Computer Conference*, volume 48, pages 313–317, June 1979.

[11] E. Oran Brigham. *The Fast Fourier Transform and Its Applications.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[12] David Cash, Alptekin Küpçü, and Daniel Wichs. Dynamic proofs of retrievability via oblivious ram. *Journal of Cryptology*, 30(1):22–57, 2017.

[13] S. Chen, L. Bai, Y. Chen, H. Jiang, and K. C. Li. Deploying scalable and secure secret sharing with gpu many-core architecture. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 1360–1369, May 2012.

[14] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, SFCS '85, pages 383–395, Washington, DC, USA, 1985. IEEE Computer Society.

[15] CNBC - News. Accessed: 01.04. 2017. `http://www.cnbc.com/2017/02/28/amazons-web-servers-are-down-and-its-causing-trouble-across-the-internet.html/`.

[16] S. A. Cookr. On the minimum computation time of functions. Master's thesis, Haward University, 1966.

[17] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[18] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael, 1999.

[19] Joan Daemen and Vincent Rijmen. *The Block Cipher Rijndael*, pages 277–284. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[20] J. L. Danger, S. Guilley, and P. Hoogvorst. High speed true random number generator based on open loop structures in fpgas. *Microelectron. J.*, 40(11):1650–1656, November 2009.

[21] D. Demirel, S. Krenn, T. Lorünser, and G. Traverso. Efficient and privacy preserving third party auditing for a distributed storage system. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pages 88–97, Aug 2016.

[22] H. S. Deshpande, K. J. Karande, and A. O. Mulani. Efficient implementation of aes algorithm on fpga. In *2014 International Conference on Communication and Signal Processing*, pages 1895–1899, April 2014.

[23] Markus Dichtl and Jovan Dj. Golić. *High-Speed True Random Number Generation with Logic Gates Only*, pages 45–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[24] Digital-Scur - Secret Sharing Library GFShare. Accessed: 01.04. 2017. `http://www.digital-scurf.org/software/libgfshare`.

[25] Morris Dworkin. Recommendation for Block Cipher Modes of Operation, Methods and Techniques . NIST Special Publication 800-38A, National Institute of Standards and Technology, 2001.

[26] EECS - GF(2m) C-library from James S. Plank. Accessed: 01.04. 2017. `http://web.eecs.utk.edu/~plank/plank/papers/CS-07-593/e`.

[27] EthernetFMC. Accessed: 01.04. 2017. `http://ethernetfmc.com/`.

[28] Can Eyupoglu. Performance analysis of karatsuba multiplication algorithm for different bit lengths. *Procedia - Social and Behavioral Sciences*, 195:1860 – 1864, 2015.

[29] X. Fang and L. Li. On karatsuba multiplication algorithm. In *The First International Symposium on Data, Privacy, and E-Commerce (ISDPE 2007)*, pages 274–276, Nov 2007.

[30] Ahmad Farooq and Khan Hamid. An efficient and simple algorithm for matrix inversion. *Int. J. Technol. Diffus.*, 1(1):20–27, January 2010.

[31] V. Fischer, A. Aubert, F. Bernard, B. Valtchanov, J. l. Danger, and N. Bochard. Project anr- icter true random number generators in configurable logic devices version 1.02, 2009.

[32] V. Fischer and F. Bernard. *True Random Number Generators in FPGAs*, pages 101–135. Springer Netherlands, Dordrecht, 2011.

[33] Viktor Fischer and Miloš Drutarovský. *True Random Number Generator Embedded in Reconfigurable Hardware*, pages 415–430. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[34] Matthias Fitzi, Juan Garay, Shyamnath Gollakota, C. Pandu Rangan, and Kannan Srinathan. *Round-Optimal and Efficient Verifiable Secret Sharing*, pages 329–342. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[35] Rosario Gennaro, Yuval Ishai, Eyal Kushilevitz, and Tal Rabin. The round complexity of verifiable secret sharing and secure multicast. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, STOC '01, pages 580–589, New York, NY, USA, 2001. ACM.

[36] Github - AES. Accessed: 01.04. 2017. `https://github.com/kokke/tiny-AES128-C/blob/master/aes.c`.

[37] Jovan Dj. Golic. New paradigms for digital generation and post-processing of random data, 2004. golic@inwind.it 12689 received 28 Sep 2004.

[38] Rogelio Adrian Hernandez-Becerril, Ariana Guadalupe Bucio-Ramirez, Mariko Nakano-Miyatake, Hector Perez-Meana, and Marco Pedro Ramirez-Tachiquin. A gpu implementation of secret sharing scheme based on cellular automata. *The Journal of Supercomputing*, 72(4):1291–1311, 2016.

[39] Ari Juels and Burton S. Kaliski, Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 584–597, New York, NY, USA, 2007. ACM.

[40] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. In *Soviet Physics - Doklady*, volume 7, pages 595–596, January 1963.

[41] Mahmoud Khasawneh, Izadeen Kajman, Rashed Alkhudaidy, and Anwar Althubyani. *A Survey on Wi-Fi Protocols: WPA and WPA2*, pages 496–511. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[42] Martin Kirchner. On the applicability of secret sharing cryptography in secure cloud services. Master's thesis, Technische Universität Wien, 2014.

[43] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[44] Paul Kohlbrenner and Kris Gaj. An embedded true random number generator for fpgas. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, FPGA '04, pages 71–78, New York, NY, USA, 2004. ACM.

[45] Hugo Krawczyk. Distributed fingerprints and secure information dispersal. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 207–218, New York, NY, USA, 1993. ACM.

[46] Hugo Krawczyk. Secret sharing made short. In *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '93, pages 136–146, London, UK, UK, 1994. Springer-Verlag.

[47] M. Kumar and A. Singhal. Efficient implementation of advanced encryption standard (aes) for arm based platforms. In *2012 1st International Conference on Recent Advances in Information Technology (RAIT)*, pages 23–27, March 2012.

[48] Rudolf Lidl and Harald Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, New York, NY, USA, 1986.

[49] T. Loruenser, A. Happe, and D. Slamanig. Archistar: Towards secure and robust cloud based data sharing. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 371–378, Nov 2015.

[50] P. Luo, A. Y. L. Lin, Z. Wang, and M. Karpovsky. Hardware implementation of secure shamir's secret sharing scheme. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 193–200, Jan 2014.

[51] Mehrdad Majzoobi, Farinaz Koushanfar, and Srinivas Devadas. *FPGA-Based True Random Number Generation Using Circuit Metastability with Adaptive Feedback Control*, pages 17–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[52] G. Marsaglia. Diehard Battery of Tests of Randomness. Florida State University, 1995.

[53] R. J. McEliece and D. V. Sarwate. On sharing secrets and reed-solomon codes. *Commun. ACM*, 24(9):583–584, September 1981.

[54] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Advanced Encryption Standard*. Alpha Press, 2009.

[55] Opencores - AES. Accessed: 01.04. 2017. `https://opencores.org/project,tiny_aes`.

[56] M. O. Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 403–409, Nov 1983.

[57] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 73–85, New York, NY, USA, 1989. ACM.

[58] L. M. Reyneri, D. Del Corso, and B. Sacco. Oscillatory metastability in homogeneous and

inhomogeneous flip-flops. *IEEE Journal of Solid-State Circuits*, 25(1):254–264, Feb 1990.

[59] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications . NIST Special Publication 800-22, National Institute of Standards and Technology, 2000.

[60] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson, Tadayoshi Kohno, and Mike Stay. The twofish team's final comments on aes selection. 2000.

[61] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3):281–292, 1971.

[62] G. Seroussi and Hewlett-Packard Laboratories. *Table of Low-weight Binary Irreducible Polynomials*. HP Laboratories technical report. Hewlett Packard Laboratories, 1998.

[63] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '08, pages 90–107, Berlin, Heidelberg, 2008. Springer-Verlag.

[64] Mehul A. Shah, Mary Baker, Jeffrey C. Mogul, and Ram Swaminathan. Auditing to keep online storage services honest. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS'07, pages 11:1–11:6, Berkeley, CA, USA, 2007. USENIX Association.

[65] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[66] Prassanna Shanmuga Sundaram. Development of a fpga-based true random number generator for space applications, 2010.

[67] D. Slamanig and C. Hanser. On cloud storage and the cloud of clouds approach. In *2012 International Conference for Internet Technology and Secured Transactions*, pages 649–655, Dec 2012.

[68] Josef Spillner, Gerd Bombach, Steffen Matthischke, Johannes Muller, Rico Tzschichholz, and Alexander Schill. Information dispersion over redundant arrays of optimal cloud storage for desktop users. In *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, UCC '11, pages 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[69] B. Sunar, W. J. Martin, and D. R. Stinson. A provably secure true random number generator with built-in tolerance to active attacks. *IEEE Transactions on Computers*, 56(1):109–119, Jan 2007.

[70] Martin Tompa and Heather Woll. How to share a secret with cheaters. *Journal of Cryptology*, 1(3):133–138, 1989.

[71] A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Physics - Doklady*, volume 7, pages 714–716, January 1963.

[72] K. H. Tsoi, K. H. Leung, and P. H. W. Leong. Compact fpga-based true and pseudo random number generators. In *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.*, pages 51–61, April 2003.

[73] Michal Varchola. Fpga based true random number generators for embedded cryptographic applications, 2008.

[74] Ihor Vasyltsov, Eduard Hambardzumyan, Young-Sik Kim, and Bohdan Karpinskyy. *Fast Digital TRNG Based on Metastable Ring Oscillator*, pages 164–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[75] Joachim von zur Gathen and Jamshid Shokrollahi. *Efficient FPGA-Based Karatsuba Multipliers for Polynomials over F2*, pages 359–369. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[76] K. Wold and C. H. Tan. Analysis and enhancement of random number generator in fpga based on oscillator rings. In *2008 International Conference on Reconfigurable Computing and FPGAs*, pages 385–390, Dec 2008.

[77] Johannes Wolkerstorfer. Secret-sharing hardware improves the privacy of network monitoring. In *Proceedings of the 5th International Workshop on Data Privacy Management, and 3rd International Conference on Autonomous Spontaneous Security*, DPM'10/SETOP'10, pages 51–63, Berlin, Heidelberg, 2011. Springer-Verlag.

[78] Xilinx - AXI 1G/2.5G Ethernet Subsystem. Accessed: 01.04. 2017. `https://www.xilinx.com/products/intellectual-property/axi_ethernet.html`.

[79] Xilinx - Vivado. Accessed: 01.04. 2017. `https://www.xilinx.com/products/design-tools/vivado.html`.

[80] Xilinx - Zynq 7000 Overview. Accessed: 01.04. 2017. `https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf`.

[81] Zhanqi Xu, Kechu Yi, and Zengji Liu. A universal algorithm for parallel crc computation and its implementation. *Journal of Electronics (China)*, 23(4):528–531, 2006.

[82] Zedboard. Accessed: 01.04. 2017. `http://zedboard.org/`.

[83] D. Zuras. On squaring and multiplying large integers. In *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, pages 260–271, Jun 1993.

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, am 25.04.2017 ______________________________

[Jakob Stangl]