David Johannes Schrammel, BSc

Low-Resource ASIC Implementation of a Lattice-Based Encryption Scheme

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Software Development and Business Management

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Stefan Mangard

Institute of Applied Information Processing and Communications (IAIK)

Graz, November 2018

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

Date

Signature

Acknowledgements

I would like to thank my thesis advisors Peter Peßl and Stefan Mangard for their continuous support and help throughout my thesis. They proposed the topic of this thesis, gave valuable suggestions, and provided guidance throughout this thesis.

I would also like to thank all other members of the Institute of Applied Information Processing and Communications (IAIK) for providing the infrastructure for the practical part and their continuous support.

Finally, I would like to thank my family for their unfailing support and encouragement throughout my years of study. This thesis would not have been possible without them.

Thank you.

Abstract

In the past few decades, quantum computers have become a reality and threaten to break currently used asymmetrical cryptographic schemes. In the same timeframe, research in quantum-secure cryptograpy has seen a rise in popularity. Today many different proposals exist, and some schemes have already seen real-world evaluations. There also already exist many efficient implementations for software and Field Programmable Gate Arrays (FPGAs). However, these implementations are not suitable for constrained devices like Near-field communication (NFC) tags, and are often geared towards one out of the many proposed parameter sets.

This thesis aims at improving the situation by presenting the first Application-Specific Integrated Circuit (ASIC) implementation of lattice-based encryption. The implementation supports many different parameter sets for two different schemes: a Chosen-Plaintext Attack (CPA)-secure encryption scheme and an Adaptive Chosen-Ciphertext Attack (CCA2)-secure encryption scheme.

Designing an ASIC implementation for constrained devices presents unique challenges, as the area and power requirements have to be kept at a minimum. On top of that, many design choices used for the FPGA implementations are not suitable for a low-resource ASIC implementation. To still achieve good performance, this implementation uses several state-of-the-art optimizations, like integrating the scaling within the Number Theoretic Transform (NTT) operation and precomputing the so-called twiddle factors. For hashing, an existing implementation of the Keccak (SHA-3) algorithm was integrated. For the generation of error polynomials, a binomial sampler was implemented instead of using a Gaussian sampler. This significantly reduces the complexity and area requirements. Trivium was integrated to act as a Pseudorandom Number Generator (PRNG) for the sampler. For the main memory, a single-port static RAM (SRAM) macro is used. To counter side-channel attacks, the entire implementation runs in constant and data-independent time.

The implementation was synthesized using a UMC 65nm manufacturing process. The area requirements are 27.4kGE (39 558μm^2), from which the RAM uses 14.7kGE (21 242μm^2). When instantiating without support for CCA2 security, the area requirements are as low as 19.5kGE (28 100μm^2). Depending on the used parameter set, a CPA-secure encryption takes between 32 908 and 143 335 cycles, a decryption, between 12 168 and 55 842 cycles. For the CCA2-secure scheme, especially decryptions take longer, with up to 251 063 cycles. At the maximum operating frequency of 70MHz encryptions take as little as as 470μs. The power consumption is less than 47μW/MHz, which is suitable for passively-powered devices like NFC tags.

Keywords: Lattice-based Cryptography, Number Theoretic Transform, Post-Quantum Cryptography Application-Specific Integrated Circuit, Ring-Learning With Errors

Kurzfassung

In den letzten Jahrzehnten wurden Quantencomputer zur Realität und drohen weit verbreitete asymmetrische Verschlüsselungsverfahren zu brechen. Gleichzeitig erfreute sich die Forschung an Post-Quanten Verschlüsselungsverfahren immer größerer Beliebtheit. Es existieren heute schon viele verschiedene solcher Verfahren, wovon manche auch schon in der Praxis getestet wurden. Weiters gibt es bereits viele effiziente Implementierungen für Software und Field Programmable Gate Arrays (FPGAs), jedoch sind diese für sehr resourcenlimitierte Geräte wie Near-field communication (NFC)-Tags nicht geeignet. Außerdem sind diese Implementierungen meistens nur für ein bestimmtes Parameter-Set optimiert.

Diese Arbeit präsentiert die erste Application-Specific Integrated Circuit (ASIC) Implementierung eines gitterbasierten, quantencomputerresistenten, asymmetrischen kryptographischen Verfahrens und schließt damit diese Lücke. Die Implementierung unterstützt viele verschiedene Parameterisierungen für zwei verschiedene Verschlüsselungsferfahren: das erste bietet Sicherheit gegen Chosen-Plaintext-Angriffen (CPA), das zweite auch gegen Adaptive-Chosen-Ciphertext-Angriffen (CCA2). Das Entwerfen einer ASIC-Implementierung für limitierte Geräte stellt eine besondere Herausforderung dar, beispielsweise müssen die Fläche sowie der Stromverbrauch minimal gehalten werden. Außerdem können viele Designentscheidungen vorhandener FPGA-Implementierungen für eine ressourcenschonende ASIC-Implementierung nicht übernommen werden. Um trotzdem eine gute Performance zu erzielen, werden viele Optimierungen verwendet. Als Hashfunktion wurde eine existierende Implementierung des Keccak (SHA-3) Algorithmus integriert. Für die Generierung von Fehlerpolynomen wird eine Binomialverteilung verwendet. Der dafür benötigte Zufallszahlengenerator wurde mit Trivium realisiert. Für den Hauptspeicher wird ein Statischer RAM (SRAM) mit einem Port verwendet. Um Seitenkanalattacken entgegenzuwirken, läuft die gesamte Implementierung in datenunabhängiger und konstanter Zeit.

Das Design wurde für einen 65nm Herstellungsprozess von UMC synthetisiert. Die resultierenden Flächenanforderungen betragen 27.4kGE (39 558μm^2), wovon 14.7kGE (21 242μm^2) für den RAM benutzt werden. Ohne Unterstützung für das CCA2-sichere Verfahren werden nur 19.5kGE (28 100μm^2) benötigt. Eine CPA-sichere Verschlüsselung beansprucht zwischen 32 908 und 143 335 Zyklen und eine Entschlüsselung 12 168 bis 55 842 Zyklen. Für das CCA2-sichere Verfahren werden bis zu 251 063 Zyklen benötigt. Mit der maximalen Taktfrequenz von 70MHz brauchen Verschlüsselungen nur 470μs. Der Stromverbrauch ist dabei unter 47μW/MHz. Somit eignet sich diese Implementierung auch für passiv betriebene Geräte wie NFC Tags.

Stichwörter: Gitterbasierte Kryptographie, Number Theoretic Transform, Post-Quanten-Kryptographie, Anwendungsspezifische integrierte Schaltung, Ring-Learning With Errors

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Currently used asymmetric encryption schemes are based on hard mathematical problems. The ideas on which they are built upon date back several decades. The RSA encryption algorithm was published in 1977, Elliptic Curve Cryptography (ECC) was proposed in 1985 but has not entered widespread use until 2004.

In 1994, Peter Shor proposed an algorithm that can break these popular cryptographic systems in polynomial time [74]. However, it requires the use of a large quantum computer, which was not available at that time. Since then, quantum computers have improved steadily. Currently available quantum computers are still very limited in performance and functionality, but they show great potential [55, 28]. It is not clear if or when quantum computers will be capable of performing Shor's algorithm efficiently. Their possible advent, however, led to question the security of currently used encryption schemes. For instance, in 2016 the National Security Agency (NSA) advised transitioning away from traditional asymmetric schemes like RSA and ECC [49].

In recent years, quantum-resistant cryptography has become a very popular research topic. Today there already exist many different proposals for quantum-resistant schemes for signatures [31, 23, 50], public-key encryption [46, 14] and many more applications [75, 12, 10, 4, 25].

While no official standard exists yet, many of these proposals have been tested, and some are used in practice. For instance, Google has previously been testing quantum-secure key-exchange schemes in their Web-Browser Chrome in 2016 [13]. The most promising candidates for quantum-secure schemes are lattice-based cryptography, code-based cryptography, supersingular elliptic curve cryptography, or hash-based cryptography.

Out of these, especially lattices have proven to be highly practical. Many modern lattice-based schemes rely on the Ring-LWE problem, which is a hard mathematical problem that cannot be broken using Shor's algorithm. It allows the construction of very efficient schemes. While there already exist several efficient implementations of Ring-LWE-based schemes for software [31, 20, 44] and Field Programmable Gate Arrays (FPGAs) [71, 66, 64, 69, 16], they are not suitable for use in constrained devices, and they typically do not feature any resistance to side-channel attacks. Additionally, many implementations are optimized towards a specific parameter set, which is not future-proof since there exists no standard yet.

This thesis aims at improving the situation by presenting the first Application-Specific Integrated Circuit (ASIC) implementation of a lattice-based encryption scheme. This implementation focuses on low-resource use, will support many different parameter sets, and feature some resistance to side-channel attacks. This allows for a fair evaluation and comparison with state of the art hardware implementations of other asymmetric encryption schemes, like RSA and ECC.

Organization of this Thesis

This thesis is organized as follows. In Chapter 2, the mathematical background for lattices is given and lattice problems, which form the basis of lattice-based cryptography, are introduced. Then, a definition of an efficient Ring-Learning-with-Errors-based encryption scheme is given. Afterwards, different algorithms for sampling from Gaussian distributions are explored, which are used in state of the art implementations of lattice-based cryptography. Then, an improved scheme is presented, which is secure against adaptive-chosen-ciphertext attacks. Finally, an overview of current hardware implementations of lattice-based cryptography is given.

In Chapter 3, the goals and requirements of the implementation are defined, and basic design choices are discussed. Finally, an overview of the design is given. Following that, Chapter 4 will give detailed descriptions of the design and algorithms used.

In Chapter 5, the circuit size, runtime, and power consumption of the implementation are analyzed and compared with efficient hardware implementations of traditional public-key encryption schemes.

Finally, Chapter 6 summarizes and discusses the results of this thesis and provides an outlook for future work.

Chapter 2

Lattice-Based Cryptography

This chapter gives an overview of the most important aspects of lattice-based cryptography and describes an efficient lattice-based encryption scheme. The presented scheme is a candidate for future post-quantum-secure asymmetric encryption schemes and is based on a mathematical structure called *lattice*. Lattices are used to build a variety of cryptographic schemes because they provide useful properties that current cryptographic schemes do not provide.

Section 2.1 introduces the concept of lattices. Section 2.2 gives an overview of the current state-of-the-art in the field of lattice-based cryptography. The security of lattice-based cryptographic schemes relies on the hardness of certain problems that are defined on lattices. However, while these lattice problems are not directly used to build cryptographic schemes, there exist related problems that can be used for building cryptographic schemes. Section 2.3 presents both kinds of these problems. Cryptographic schemes that are based on regular lattices require large keys and have a high runtime. So-called *ideal* lattices have been proposed to avoid these problems. Ideal lattices feature some additional structure which can be used to build more efficient cryptographic schemes. Section 2.4 introduces the concept of ideal lattices and show how using a problem defined on ideal lattices can help building a more efficient encryption scheme. Section 2.5 first presents the Number Theoretic Transform (NTT), which is similar to a Fast Fourier Transform (FFT) and can be used for efficient polynomial multiplications. Then, based on the NTT, a more efficient encryption scheme is presented. Many lattice-based cryptographic schemes require random samples which are sourced from a discrete Gaussian distribution. In Section 2.6, an overview of the state-of-the-art in the design of discrete Gaussian samplers is given, and binomial distributions as an approximation to discrete Gaussian distributions are introduced. Section 2.7 introduces a CCA2-conversion of the previously defined efficient encryption scheme to make it secure against chosen-ciphertext attacks. Both of the presented encryption schemes are implemented in the practical part of this thesis. Finally, Section 2.8 discusses existing hardware implementations related to lattice-based cryptography.

2.1 Lattices

Definition 2.1.1. A lattice $\mathcal{L} \in \mathbb{R}^n$ is a set of points in n-dimensional space with a periodic structure. Each point in the lattice can be described by an integer combination of the n linearly independent basis vectors $\mathbf{a}_1, \ldots, \mathbf{a}_n \in \mathbb{R}^n$ that fully determine the lattice:

$$\mathcal{L}\left(\mathbf{a}_1, ..., \mathbf{a}_n\right) = \left\{ \sum_{i=1}^{n} x_i \mathbf{a}_i : x_i \in \mathbb{Z} \right\}$$

Figure 2.1 shows an example of a two-dimensional lattice. Any point in the lattice can be reached by an integer combination of either black or red vector bases. There exists an infinite number of possible vector bases for any lattice. For cryptographic applications, *q-ary* lattices are of particular interest. Such *q-ary* lattices are lattices that contain a vector $\mathbf{x}$ if and only if $\mathbf{x}$ *mod* q is also in the lattice. Unless stated otherwise all lattices in this thesis are assumed to be *q-ary*.

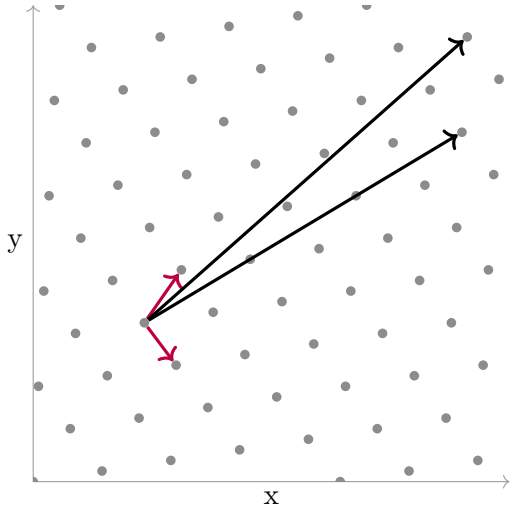


Figure 2.1: A two-dimensional lattice with two possible vector bases (red and black).

2.2 Lattice-Based Cryptography

The term lattice-based cryptography is used for cryptographic constructions that are based on lattice problems. Lattices have been studied for a long time, in the late 18th century Lagrange [38] and Gauss [34] have already discovered the hardness of certain lattice problems. The first significant breakthrough in lattice-based cryptography however happened in 1996, when Ajtai proposed a public-key scheme based on lattices [2].

Lattice-based cryptography promises provable security, resistance against quantum computers, as well as worst-case hardness. Currently, there exists no widely used quantum computer resistant asymmetric encryption scheme, so the quantum computer resistance is a very beneficial property.

Quantum computing was shown to break RSA or ECC-based schemes in polynomial time [74]. It is unknown if and when quantum computers become powerful enough to be a considerable threat to asymmetric cryptographic schemes which are currently in use. However, there is a continuous effort and progress in construction of better quantum computers [19, 54, 62]. The quantum-resistance property of lattice-based cryptographic schemes, as well as increased practicality of quantum computers have led to an increase of activity in this field of research in the past decade.

Thanks to this research, there have been numerous proposals of lattice-based schemes for many different applications like hash functions [45], signatures [31], and public-key encryption [46]. All lattice-based schemes are built on top of the several lattice-related problems which are assumed to be hard to solve even for quantum computers. In other words, there is no quantum or classical algorithm known for solving these problems efficiently. These lattice-related problems are not always directly defined on lattices but can be reduced to real lattice problems. In the next section, these lattice problems as well as lattice-related problems, are presented.

2.3 Lattice Problems

This section presents the two most important lattice problems (Shortest Vector Problem and Closest Vector Problem), which play a significant role in the provable security of lattice-based cryptography. This security is based on the presumption that these problems are hard to solve. Additionally, the Learning With Errors Problem (LWE) is presented. It is not directly defined on a lattice, but it can be reduced to a lattice problem.

Notation. In this work, lower-case bold symbols are used to indicate that a variable $\mathbf{v}$ is a vector or polynomial. Upper-case bold symbols indicate that a variable $\mathbf{A}$ is a matrix. $\langle \mathbf{v}, \mathbf{w} \rangle$ denotes a dot product (scalar product) of the two vectors $\mathbf{v}$ and $\mathbf{w}$.

2.3.1 Shortest Vector Problem

Definition 2.3.1. Given a lattice $\mathcal{L}$ which is defined by n linearly independent and uniformly random basis vectors, the Shortest Vector Problem (SVP) is defined as finding a vector $\mathbf{x} \in \mathcal{L}$ with a length equal to the shortest vector in $\mathcal{L}$. In other words, the goal is to find the shortest non-zero vector in $\mathcal{L}$.

Usually, the SVP does not require the solver to find exactly the shortest vector, but only a short vector which is smaller than the length of the true shortest vector multiplied with an approximation factor γ. This γ-approximation to the SVP is often denoted as SVP_γ. Figure 2.2 gives a visual representation of the SVP using a two-dimensional lattice.

The SVP is easy to solve if the basis vectors are already close to the shortest vector. However, if the basis vectors are chosen randomly, the probability for an easy problem is negligible for lattices with a high dimension. SVP was shown to be NP-hard, there exist algorithms to solve the SVP, but they require exponential time [72].

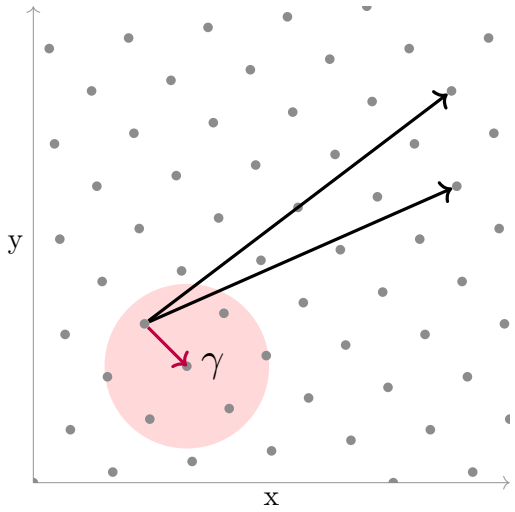


Figure 2.2: Shortest Vectors Problem: Given a Lattice $\mathcal{L}$ with a vector base (**black**), find the shortest vector (**red**) or a γ-approximation (SVP_γ).

2.3.2 Closest Vector Problem

Definition 2.3.2. Given a lattice $\mathcal{L}$ and a target vector $\mathbf{t}$ (which is not on the lattice), the Closest Vector Problem (CVP) is defined as finding a vector $\mathbf{x} \in \mathcal{L}$ which is closest to the target vector $\mathbf{t}$.

For this problem, there also exists a γ-approximation where it is not required to find the exact solution, but only a vector, such that the distance to the target vector is at most a polynomial factor γ longer than the true distance. This γ-approximation to the CVP is often denoted as CVP_γ. Both problems, both SVP_γ and CVP_γ, correspond strongly with each other, as there exist reductions in both directions [47].

A visualization of the CVP problem for a two-dimensional lattice is given in Figure 2.3.

2.3.3 Learning with Errors Problem

The Learning With Errors Problem (LWE) was introduced by Regev in 2005 together with an efficient cryptosystem based on the hardness of this problem [68].

Definition 2.3.3. Given a lattice $\mathcal{L}$ defined by n linearly independent basis vectors $\mathbf{a}_i$ that are uniformly random, an error distribution $\mathcal{X}$ (typically a discrete Gaussian), a modulus q, and an arbitrary number j of tuples with the following structure:

$$\begin{aligned}
(\mathbf{a}_{i_1} \ , \ b_1) &= (\mathbf{a}_{i_1} \ , \ \langle \mathbf{a}_{i_1}, \mathbf{s} \rangle + e_1) \mod q \\
&\vdots \\
(\mathbf{a}_{i_j} \ , \ b_j) &= (\mathbf{a}_{i_j} \ , \ \langle \mathbf{a}_{i_j}, \mathbf{s} \rangle + e_j) \mod q,
\end{aligned}$$

where the error term $e_x \in \mathbb{Z}_q$ is sampled from $\mathcal{X}$, and a secret $\mathbf{s} \in \mathbb{Z}_q^n$ is chosen uniformly at random, the search variant of the LWE problem requires one to find the secret $\mathbf{s}$. The coefficient-wise vector multiplication $b_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i$ can also be rewritten as a matrix-vector multiplication $\mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e}$, where $\mathbf{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_n]$.

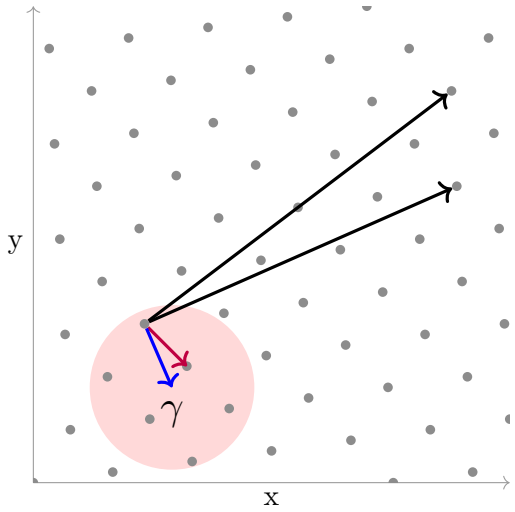


Figure 2.3: Closest Vector Problem: Given a vector base (**black**) and a target vector (**blue**), find the vector closest to the target vector (**red**) or a γ-approximation (CVP_γ).

There also exists a decision variant of the LWE problem which asks the solver to distinguish between an arbitrary number of LWE-tuples from the lattice $\mathcal{L}$ and tuples $(\mathbf{a}_i \ , \ b')$ where b' is chosen uniformly at random. The underlying assumption is that even though $b_1, \ldots, b_j$ are not distributed uniformly random, the attacker is unable to distinguish between them and truly uniformly random values b'.

By reducing the LWE problem to a variant of the SVP_γ problem, which is proven to be a hard problem, Regev [68] has shown that both the decision variant and the search variant are equivalent and at least as hard as worst-case lattice problems. This proof has been confirmed and improved upon by Peikert [56]. Even though both LWE problems are equally hard, only the decision variant is currently used for proving the security of cryptographic schemes.

The LWE problem turned out to be very versatile and useful for cryptographic applications. It has been proposed to be used for public-key encryption schemes [68], oblivious transfer protocols [59], leakage-resilient encryption [3, 5, 21], identity-based encryption [27, 81, 15, 1], as well as for fully homomorphic encryption [26].

Currently proposed public-key encryption schemes based on the standard LWE problem [68] have significant drawbacks compared to traditional schemes like ECC and RSA. The lattice basis has to be stored in the key, so the key sizes are much bigger and too impractical for use in typical web connection handshakes. The runtime cost is also much larger due to the vector operations, that are needed for every bit of the ciphertext. Thankfully there exists a more efficient variant called Ring-LWE, which will be introduced in the next section.

2.4 Ideal Lattices and Ring-Learning with Errors

This section presents ideal lattices, which can be used to improve the practicality of lattice-based cryptographic schemes. Compared to regular lattices, ideal lattices have the advantage that vector operations can be replaced with efficient polynomial multiplications.

Initial LWE encryption schemes used matrix-vector operations. However, these are very slow and have large key sizes. Lyubashevsky et al. [46] proposed a more efficient variant called Ring-LWE, which uses structured ideal lattices. These ideal lattices allow replacing all vector operations with polynomial multiplications. Thus, schemes based on Ring-LWE are typically much more efficient and have smaller key sizes.

Notation. In this thesis, $\mathbf{x} * \mathbf{y}$ is used denote point-wise multiplications and $\mathbf{x} \cdot \mathbf{y}$ to denote polynomial multiplications.

2.4.1 Ideal Lattices

Ideal lattices are lattices that provide additional structure. Basis vectors, i.e., the columns of $\mathbf{A}$, are chosen uniformly at random in a regular lattice. In ideal lattices, at least in the ones which are used in this thesis, the vectors are nega-cyclic shifts of one another, which means that each vector is an element-wise rotation of the previous vector with the negation of the first element.

Compared to regular lattices, the space requirements for ideal lattices are reduced from $\mathcal{O}(n^2)$ to just $\mathcal{O}(n)$. Ideal lattices can be interpreted as ideals in a specific finite ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(f)$, where f is an irreducible polynomial of degree n and required to be monic. For efficiency reasons, f is usually chosen to be $x^n + 1$, where n is a power of two and q is chosen as a prime satisfying $q \equiv 1 \mod 2n$.

An example of such a basis vector of an ideal lattice is shown in Figure 2.4.

1	2	3	-4
4	1	2	3
-3	4	1	2
-2	-3	4	1

Figure 2.4: Illustration of the basis of an an ideal lattice with $f = x^n + 1$ and its nega-cyclic shift property. Each row is the result of the previous row shifted to the right with the first element negated.

Using ideal lattices reduces memory requirements significantly because they can be defined using a single vector. They also offer significant time savings, because all matrix multiplications can be replaced with polynomial multiplications which are much more efficient. Multiplications still happen in $\mathcal{R}_q$, which means that they are still time-consuming due to the reduction steps. However, the multiplication can be optimized by using the NTT algorithm, which will be described in Section 2.5.

2.4.2 Ring-Learning with Errors Problem

Definition 2.4.1. Given an ideal lattice, an error distribution (typically a Gaussian), the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(f)$, a modulus q and samples $(\mathbf{a}_1, \mathbf{t}_1), \ldots, (\mathbf{a}_m, \mathbf{t}_m) \in \mathcal{R}_q \times \mathcal{R}_q$, an attacker has to distinguish randomly chosen samples from samples $\mathbf{t}_i = \mathbf{a}_i\mathbf{s} + \mathbf{e}_i$ where $\mathbf{s}$ is a uniformly sampled secret value and $\mathbf{e}_1, \ldots, \mathbf{e}_m$ drawn from the error distribution.

The Ring-LWE is the ring variant of the LWE problem. It was first introduced by Lyubashevsky et al. in 2010 [46]. Like the LWE problem, the Ring-LWE problem has two variants.The search variant of the Ring-LWE problem requires an attacker to output $\mathbf{s}$. For the decision variant, the attacker has to decide if samples are chosen uniformly at random or if they are valid Ring-LWE-tuples in the form $\mathbf{t}_i = \mathbf{a}_i\mathbf{s} + \mathbf{e}_i$.

In standard LWE schemes, each sample is defined as $\mathbf{b}'_x = \langle \mathbf{a}_x, \mathbf{s} \rangle + \mathbf{e}_x$, but this is inefficient because it requires the computation of a dot product for each sample. In Ring-LWE, dot multiplications can be replaced by polynomial multiplications in a finite ring. Thus n samples can be generated at once by calculating $\mathbf{b} = \mathbf{a} \cdot \mathbf{s} + \mathbf{e} \in R_q$.

The Ring-LWE problem has had a significant impact on the construction of practical lattice-based encryption schemes. LWE-based encryption schemes usually come with a security proof that ensures NP-hardness, however, for Ring-LWE no such classical reduction to an NP-hard problem exists. The only known reduction to an NP-hard problem is quantum [46]. Thus, it requires a quantum computer in order to be efficient.

2.4.3 Ring-LWE Encryption Scheme

Initial LWE encryption systems were based on matrix-vector operations. They were inefficient and used large key sizes. This section describes a more efficient encryption scheme based on ideal lattices and the Ring-LWE problem. It was first proposed in 2010 by Lyubashevsky et al. [46, 41].

The scheme is defined in $\mathbb{Z}_q[x]/(x^n + 1)$ and parameterized by the probability distributions $\mathcal{U}$ and $\mathcal{X}_\sigma$, a prime q, as well as an encoder- and decoder-function. $\mathcal{U}$ is a uniform distribution, and $\mathcal{X}_\sigma$ is a discrete Gaussian distribution with a zero mean and a standard deviation of σ. Additionally, an n-dimensional ideal lattice, which is defined by the polynomial $\mathbf{a} \in R_q$ is required. This variable $\mathbf{a}$ is sampled from $\mathcal{U}$ and is a part of the public key.

The encoder function maps an n-bit input vector $\mathbf{m}$ to $\bar{\mathbf{m}} \in R_q$ by multiplying each bit with $\frac{q}{2}$. The decoder function maps a vector $\bar{\mathbf{m}} \in R_q$ to a binary vector by mapping values in the interval $(-\frac{q}{4}, \frac{q}{4}\ ]$ to '0' and others to '1'. A visual representation of the encoder- and decoder functions is given in Figure 2.5.

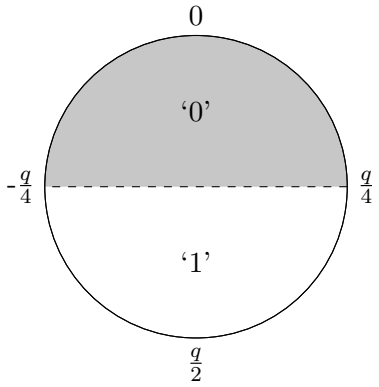


Figure 2.5: The decoder function maps coefficients from a polynomial $\in R_q$ to binary values: Values in the interval $(-\frac{q}{4}, \frac{q}{4}\ ]$ (the shaded area) get mapped to ‘0’ and all other values to ‘1’. In other words, values that are near 0 get mapped to ‘0’, and values near $\frac{q}{2}$ get mapped to ‘1’. The encoder function maps binary values ‘0’ to $0 \in \mathbb{Z}_q$ and ‘1’ to $\frac{q}{2} \in \mathbb{Z}_q$.

Key generation, encryption, and decryption are defined as follows:

- **KeyGen($\mathbf{a}$):** Sample two polynomials $\mathbf{r}_1, \mathbf{r}_2 \in R_q$ from $\mathcal{X}$ and then calculate $\mathbf{p} = \mathbf{r}_1 - \mathbf{a} \cdot \mathbf{r}_2$. The polynomial $\mathbf{r}_1$ is no longer required after key generation. The output of this function is the secret key $\mathbf{r}_2$ and public key $(\mathbf{a}, \mathbf{p})$.
- **Encrypt($\mathbf{a}, \mathbf{p}, \mathbf{m}$):** Encode the message $\mathbf{m}$ to $\bar{\mathbf{m}} \in R_q$ using the encoder function. Sample three error polynomials $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3 \in R_q$ from $\mathcal{X}$. Then compute the ciphertext $(\mathbf{c}_1, \mathbf{c}_2 \in R_q)$ as:

$$\mathbf{c}_1 \leftarrow \mathbf{a} \cdot \mathbf{e}_1 + \mathbf{e}_2$$
$$\mathbf{c}_2 \leftarrow \mathbf{p} \cdot \mathbf{e}_1 + \mathbf{e}_3 + \bar{\mathbf{m}}$$

- **Decrypt($\mathbf{c_1}, \mathbf{c_2}, \mathbf{r}_2$):** Compute $\mathbf{m}' = \mathbf{c}_1 \cdot \mathbf{r}_2 + \mathbf{c}_2 \in R_q$ and recover original message $\mathbf{m}$ from $\mathbf{m}'$ by using the decoder function.

2.5 Efficient Implementations

This section describes how the above scheme can be efficiently implemented.

2.5.1 Efficient Polynomial Multiplication

The Number Theoretic Transform (NTT) is the core of many efficient lattice-based constructions. It allows for efficient polynomial multiplication with a quasi-linear runtime complexity of $\mathcal{O}(n \log n)$ (compared to $\mathcal{O}(n^2)$ using the traditional schoolbook method). The NTT is essentially an FFT with the difference that it operates only in a specific finite ring to avoid complex arithmetic or inaccurate floating-points.

Notation. A superscript tilde symbol indicates that a variable $\widetilde{\mathbf{x}}$ is the NTT transformation of $\mathbf{x}$. Furthermore, INTT and IFFT are used to indicate the inverse operation for NTT and FFT, respectively.

An n-point FFT can be used to evaluate an n-degree polynomial in the n-th root of unity and perform a polynomial multiplication $\mathbf{c} = \mathbf{a} \cdot \mathbf{b}$ by calculating $\mathbf{c} = \text{IFFT}_{\omega_n}(\ \text{FFT}_{\omega_n}(\ \mathbf{a}\ )\ *\ \text{FFT}_{\omega_n}(\mathbf{b})\ )$. Similarly, with an NTT, the roots of unity from a finite ring $\mathcal{S}_q = \mathbb{Z}_q[x]/[\cdot]$ are used. This means that the polynomial multiplication is also performed in this ring. The complex roots ω_n in an FFT are replaced with primitive n-th roots of unity in $\mathbb{Z}_q$. The values for ω are primitive n-th roots of unity if q is a prime and $\omega^n \equiv 1 \mod q$ where $\omega^m \not\equiv 1 \mod q$ for all $m < n$. These primitive n-th roots of unity enable an efficient polynomial multiplication in the ring $\mathcal{S}_q = \mathbb{Z}_q[x]/(x^n - 1)$. The multiplication $\mathbf{c} = \mathbf{a} \cdot \mathbf{b}$ in $\mathcal{S}_q$ can now be calculated with $\mathbf{c} = \text{INTT}_{\omega_n}(\ \text{NTT}_{\omega_n}(\ \mathbf{a}\ )\ *\ \text{NTT}_{\omega_n}(\mathbf{b})\ )$.

Using the NTT, which operates in the ring $\mathcal{S}_q$, for lattice schemes still needs some adaption. Almost all lattice schemes use the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$ instead of $\mathcal{S}_q = \mathbb{Z}_q[x]/(x^n - 1)$. The relation between their primitive roots can be exploited to accomplish polynomial multiplication in $\mathcal{R}_q$. This works by scaling the input by the $2n$-th primitive roots of unity and the output by the inverse exponents. To calculate $\mathbf{c} = \mathbf{a} \cdot \mathbf{b}$ in $\mathcal{R}_q$ using the NTT, it can be rewritten as follows:

$$a'_i \text{ of } \mathbf{a}' = a_i \cdot \omega_{2n}^i$$
$$b'_i \text{ of } \mathbf{b}' = b_i \cdot \omega_{2n}^i$$

$$\mathbf{c}' = \text{INTT}_{\omega_{2n}}(\ \text{NTT}_{\omega_{2n}}(\ \mathbf{a}'\ )\ *\ \text{NTT}_{\omega_{2n}}(\ \mathbf{b}'\ )\ )$$
$$c_i \text{ of } \mathbf{c} = c'_i \cdot n^{-1}\omega_{2n}^{-i}$$

The INTT works the same way, but the inverse roots of unity are used instead. An iterative description of the NTT algorithm is given in Algorithm 1. It describes the common Colley and Tukey radix-2 decimation in time approach. The *BitReverse* operation in line 1 is used to reorder the input. This operation determines the new position of an element in $\mathbf{x}$ by reversing the binary representation of its previous position. The factors ω are also called *twiddle factors*. At the core of the algorithm (lines 7–10) lies the butterfly operation, which is the multiplication of the factor ω with one element and subtracting or adding the result (t) with another element (u).

As shown in Figure 2.6, an NTT operation, like the FFT, can be implemented using a butterfly. When using more coefficients, a butterfly network with a recursive structure can be built. Such a butterfly network with four coefficients is illustrated in Figure 2.7. The left-hand side of this figure shows the input, which has already been reordered using the *BitReverse* operation, while the right-hand side shows the output of the butterfly network. By using this representation, the butterfly-operation becomes apparent as the core of the NTT algorithm.

Algorithm 1 Iterative $n-$coefficient NTT

Input:

$\mathbf{x}$	Polynomial $\in \mathbb{Z}_q^n$
ω_n	$n-$th primitive root of unity $\in \mathbb{Z}_q$

Output:

$\widetilde{\mathbf{x}}$	Polynomial $\in \mathbb{Z}_q^n = \mathrm{NTT}(\mathbf{x})$

$\widetilde{\mathbf{x}} \leftarrow \mathrm{BitReverse}(\mathbf{x})$
for $m = 2$ to n by $m = 2m$ **do**
  $\omega_m \leftarrow \omega_n^{n/m}$
  $\omega \leftarrow 1$
  for $j = 0$ to $\frac{m}{2} - 1$ **do**
    for $k = 0$ to $n - 1$ by m **do**
      $t \leftarrow \omega \cdot \widetilde{\mathbf{x}}[k + j + \frac{m}{2}]$
      $u \leftarrow \widetilde{\mathbf{x}}[k + j]$
      $\widetilde{\mathbf{x}}[k + j] \leftarrow u + t$
      $\widetilde{\mathbf{x}}[k + j + \frac{m}{2}] \leftarrow u - t$
    end for
    $\omega \leftarrow \omega \cdot \omega_m$
  end for
end for

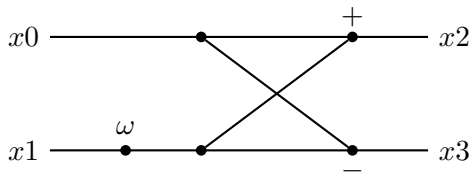


Figure 2.6: A single NTT butterfly

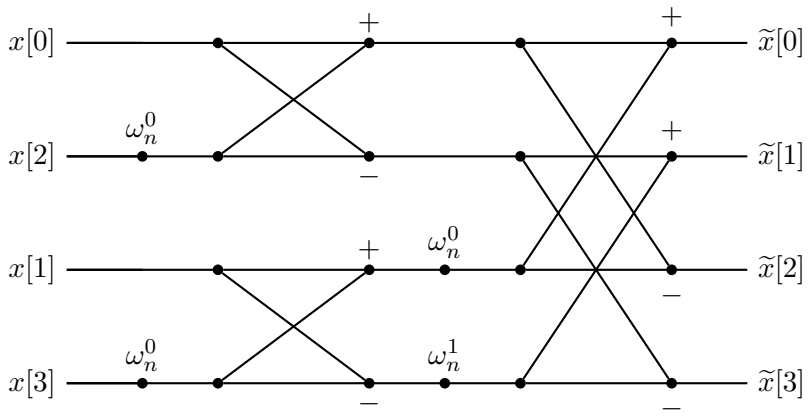


Figure 2.7: A 4-coefficient NTT butterfly network

2.5.2 Efficient Ring-LWE Encryption Scheme

The encryption scheme presented in Section 2.4.3 can be made much more efficient by utilizing the NTT operation and other optimizations. This section presents a more efficient Ring-LWE encryption scheme as implemented in the practical part of this thesis. It is much more efficient than its LWE counterpart and compared to ECC, the operations are much faster [71].

Using the NTT operation, polynomial multiplications can be computed very efficiently with a runtime of $\mathcal{O}(n \log n)$: $\mathbf{b} = \mathbf{a} \cdot \mathbf{s} + \mathbf{e} \in R_q$ can be rewritten as $\mathbf{b} = \text{INTT}(\ \text{NTT}(\mathbf{a}) * \text{NTT}(\mathbf{s}) + \text{NTT}(\mathbf{e})\ )$.

Pöppelmann and Güneysu have proposed to keep fixed polynomials, the private and public keys, in the NTT domain [64]. This reduces the number of NTT operations in *Encrypt* and *Decrypt*. Furthermore, the ciphertexts can also be stored as the NTT-transformed version as proposed by Roy et al. [71]. This allows for more efficient decryption, where only one NTT invocation is required.

Using these optimizations, the improved Ring-LWE encryption scheme is defined as follows:

- **KeyGen($\mathbf{a}$):** Sample the polynomial $\mathbf{r}_1 \in R_q$ from $\mathcal{X}$, choose a different polynomial $\mathbf{r}_2 \in R_q$ from binary coefficients and then calculate $\mathbf{p} = \mathbf{r}_1 - \mathbf{a} \cdot \mathbf{r}_2$. The polynomial $\mathbf{r}_1$ is no longer required after key generation. Perform an NTT transformation for the three polynomials $\mathbf{a}, \mathbf{p}$ and $\mathbf{r}_2$ to get $\tilde{\mathbf{a}}, \tilde{\mathbf{p}}$ and $\tilde{\mathbf{r}}_2$. The output of this function is the secret key $\tilde{\mathbf{r}}_2$ and public-key $(\tilde{\mathbf{a}}, \tilde{\mathbf{p}})$.

- **Encrypt($\tilde{\mathbf{a}}, \tilde{\mathbf{p}}, \mathbf{m}$):** Encode the message $\mathbf{m}$ to $\bar{\mathbf{m}} \in R_q$ using the encoder function. Sample three error polynomials $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3 \in R_q$ from $\mathcal{X}$. Then compute the ciphertext $(\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2)$ as:

$$\begin{aligned}
\tilde{\mathbf{e}}_1 &\leftarrow \text{NTT}(\mathbf{e}_1) \\
\tilde{\mathbf{e}}_2 &\leftarrow \text{NTT}(\mathbf{e}_2) \\
\widetilde{\mathbf{c}}_1 &\leftarrow \widetilde{\mathbf{a}} * \widetilde{\mathbf{e}}_1 + \widetilde{\mathbf{e}}_2 \\
\widetilde{\mathbf{c}}_2 &\leftarrow \widetilde{\mathbf{p}} * \widetilde{\mathbf{e}}_1 + \text{NTT}(\mathbf{e}_3 + \bar{\mathbf{m}})
\end{aligned}$$

- **Decrypt($\tilde{\mathbf{c_1}}, \tilde{\mathbf{c_2}}, \tilde{\mathbf{r}}_2$):** Compute $\mathbf{m'} = \text{INTT}(\tilde{\mathbf{c}}_1 * \tilde{\mathbf{r}}_2 + \tilde{\mathbf{c}}_2)$ and recover original message $\mathbf{m}$ from $\mathbf{m'}$ by using the decoder function.

While the polynomial $\mathbf{r}_2$ is usually sampled from $\mathcal{X}$, in this work binary coefficients are used instead. This way, the above scheme is identical to the scheme presented in [71].

The presented encryption scheme is parameterized by the three variables σ, n, and q. These parameters define the security level of the scheme. Currently, there is no standardized parameter set, but many different sets have been proposed [41, 29, 48, 52, 4]. Most of the proposed parameter sets aim for a 128-bit security level. The exact parameter sets that are used in the practical part of this thesis will be discussed in Section 3.3.

2.6 Discrete Gaussian Samplers

Lattice-based constructions often require samples from a Gaussian distribution (for example [8, 23, 41, 57, 59]). In the case of Ring-LWE, error polynomials that are sampled from a discrete Gaussian distribution are required for key generation and encryption. This section will first define a Gaussian distribution. Then it will give an overview of the current state of design of discrete Gaussian samplers, which are used in lattice-based cryptographic schemes. Binomial distributions as an alternative in Ring-LWE schemes are introduced and discussed as well.

Definition 2.6.1. The probability density function of a continuous Gaussian distribution with a standard deviation of $\sigma \in \mathbb{R}_{>0}$, with the center at $\mu \in \mathbb{R}$ is evaluated at $x \in \mathbb{R}$ is defined by

$$p_{\mu,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)}$$

Definition 2.6.2. The discrete Gaussian distribution over $\mathbb{Z}$ with the center μ at 0 is defined by

$$p_{\sigma}(x) = \frac{1}{S} e^{-x^2/(2\sigma^2)},$$

where S is a normalization factor and is approximately $\sigma\sqrt{2\pi}$.

Notation. In the literature, the definition of a discrete Gaussian distribution often uses the parameter $\sigma = s/2\pi$. This parameter defines the standard deviation of the distribution. Throughout this thesis, this notation will be used for defining a discrete Gaussian distribution.

Lattice-based cryptographic schemes rely on hard problems and security proofs. Their definitions often use discrete Gaussian distributions. However, a finite machine cannot sample from a real discrete Gaussian distribution, as it would require infinite arithmetic precision. Hence, one has to sample from a distribution which is close to a real discrete Gaussian distribution. Some previous work [40, 41, 42] argues that the statistical distance between the sampled distribution and the desired discrete Gaussian distribution needs to be less than 2^{-90} in order to still provide a large enough security margin. If the accuracy of the distribution is not good enough, then the entire cryptographic-system might become insecure because the security proofs are invalidated.

To achieve the necessary precision, storing big precomputed tables or using high-precision floating-point arithmetic is usually required. Both approaches have a huge impact on the efficiency or the area requirements, which is especially important for constrained devices.

The Gaussian sampler is usually one of the biggest components in a lattice-based scheme [77], so it is important to design and implement an algorithm which is very efficient. The parameters of the Gaussian distribution, as well as the accuracy needed, are governed by the definition of the schemes and their security proofs.

Another key aspect of Gaussian samplers is implementation security. Being a part of a cryptographic scheme, it is important that the sampler itself is secure as well. Many currently-used algorithms are susceptible to timing attacks because of their non-constant time design. This side channel information can compromise the entire scheme, as shown in [30].

In the next section, different algorithms are presented and their drawbacks regarding efficiency and security are discussed. Alternatives to Gaussian distributions in Ring-LWE schemes are introduced and discussed as well.

2.6.1 Comparison of State-Of-the-Art Implementations

Several different algorithms exist to generate samples from a discrete Gaussian distribution. This section will briefly introduce several algorithms and alternative distributions which are currently being used in lattice-based schemes and implementations. Only the binomial distribution will be explained in greater detail because of its relevance to the practical part of this thesis. Other algorithms and distributions are only briefly mentioned for the sake of completeness.

Rejection Sampling.
Rejection sampling was the first method proposed to be applied in lattice-based cryptography [27], and several optimized versions exist [22]. This algorithm does not need to store any precomputed tables. However, it can be very slow and typically requires expensive high-precision floating point arithmetic.

Inverse Transform Method.
Peikert proposed to use this method to sample discrete Gaussians. This method can be implemented either by using floating point arithmetic or using big precomputed tables for each different parameter combination. These tables are also called Cumulative Distribution Table (CDT). This method can be implemented in a very fast and efficient manner [23, 66, 39, 17] and is used in practical hardware implementations [64].

Knuth-Yao.
The Knuth-Yao algorithm [35] is a commonly used method for sampling values from a discrete Gaussian distribution. It does require a precomputed table. However, recent work has achieved much higher efficiency by using smaller tables [71, 20, 77]. The algorithm itself features data-dependent branches and thus leaks timing information. At the cost of significantly higher area requirements, Roy et al. have added random shuffling to their implementation to resist side-channel attacks [70].

Earlier work, related to Ring-LWE encryption and key-exchange schemes, typically use one of these algorithms for sampling values from a high-precision discrete Gaussian distribution. However, despite recent efficiency optimizations [71, 20, 77] and implementations that feature side-channel resistance [70], they are not optimal for hardware implementations due to their inherent side-channel leakages and efficiency issues.

The required security levels for Ring-LWE schemes can also be achieved with other distributions that are close to a discrete Gaussian distribution. Ring-LWE signature schemes usually require more accurate samplers than encryption schemes. However, even for signature schemes, efficiency can be gained by using different distributions without sacrificing security.

Alkim et al. argue that a high precision sampler, which is quite resource-intensive, is often not required and proposed using a centered binomial ψ_k as error distribution [4]. Using that error distribution results in a slight reduction in security. However, they also presented a security proof and claimed that it is good enough for most applications. They further explain that, for distributions that are close to Gaussian distributions, what matters most are entropy and the standard deviation. In practice, there exists no known attack that exploits the structure or differences in errors distributions. Replacing the Gaussian distribution with a centered binomial distribution ψ_k became common practice in more recent works [4, 52, 11].

Binomial distributions.
Binomial distributions are much more efficient to sample from because no high precision floating point arithmetic or large precomputed tables are necessary and no data-dependent branches are required. Therefore, it is easy to implement sampling in constant time, which makes it secure against timing attacks. This makes binomial distributions an ideal candidate for low-resource hardware implementations.

Samples from a binomial distribution ψ_k can be calculated by computing $\sum_{i=0}^{k-1} b_i - b'_i$ where $b_i, b'_i \in \{0, 1\}$ are $2k$ uniform independent bits. This distribution is centered, its mean is 0, and has a variance of $k/2$ (a standard deviation of $\varsigma = \sqrt{k/2}$). Using a different representation of the same formula, using the hamming weight function, is $HW(b) - HW(b')$, where $b, b' \in 0, 1^k$. This shows, that sampling from a binomial distribution is essentially the same as counting bits.

Notation. To differentiate between different distributions, σ indicates the standard deviation of a Gaussian distribution $\mathcal{X}$ and ς is used for centered binomial distributions ψ_k.

Oder et al. argued that distributions that roughly follow a discrete Gaussian, the standard deviation can be considered the most important factor when describing and comparing security levels for Ring-LWE [52]. Some other distributions that are less commonly-used in lattice-based schemes include fixed distributions [10] and non-centered binary distributions [14]. For signature schemes, uniform distributions are sometimes used [9, 31].

For this work, a binomial distribution is chosen, as it can be implemented in constant time and offer, a much-improved efficiency, which is ideal for hardware implementations. Additionally, they have already been used and tested with Ring-LWE encryption schemes in practice [4, 11, 52].

2.7 CCA2 Conversion of the Ring-LWE Encryption Scheme

Before Ring-LWE-based encryption can be seriously considered as a replacement for current encryption schemes like RSA and ECC, it has to be made secure against Adaptive Chosen-Ciphertext Attack (CCA2). Furthermore, CCA2-security is a prerequisite before any side-channel resistance can be considered because an attacker with physical access to a decryption oracle could simply use malformed ciphertexts to reveal the secret key without using a side-channel attack at all.

This section introduces an improvement to the Ring-LWE encryption scheme presented in Section 2.5.2 with added security to make it resilient against adaptive chosen-ciphertext attacks. This conversion was previously presented by Oder et al. [52]. To enable a semantically secured encryption with respect to adaptive chosen-ciphertext attacks, they use a post-quantum variant of the Fujisaki-Okamoto transformation [24] proposed by Targhi and Unruh [78]. Peikert concluded that for this transformation, a passively secured encryption scheme should be converted to an actively secured one [58]. This conclusion is based on the random oracle model, assuming adaptive attacks for CCA2.

For this transformation, the two hash functions $G : \{0,1\}^L \rightarrow \{0,1\}^l$ and $H : \{0,1\}^{L+l} \rightarrow \{0,1\}^\lambda$ are needed. Here, L is the size of the message, l the length of the input for the encryption, and λ the length of the seed for the Pseudorandom Number Generator (PRNG). For this work, L and l are set to be equal to the parameter n from the previous scheme. λ is set to 160 bits, as this is the largest seed that the used PRNG supports. Details about the PRNG will be presented in Chapter 3.

It should be noted that a third hash function is actually required for the intended quantum security. Oder et al. revised their scheme in a later version of their work [52]. However this was published after this implementation was complete, so in this work, their previous scheme is used and presented below.

Notation. To differentiate between the plain Ring-LWE encryption scheme (see Section 2.5.2), which is only secure against CPAs, and the CCA2-secure scheme, CPA_{enc} and CPA_{dec} are used to indicate a plain Ring-LWE encryption/decryption and CCA_{enc} and CCA_{dec} for the CCA2-secured variant.

The definition of the CCA2-secure encryption algorithm CCA_{enc} is given in Algorithm 2. The decryption algorithm CCA_{dec} is defined in Algorithm 3. To visualize the data-flow of the algorithms, Figure 2.8 and Figure 2.9 show a graphical representation of this CCA2-converted Ring-LWE encryption scheme.

Algorithm 2 $\mathrm{CCA}_{\mathrm{enc}}$

Input:

$(\tilde{\mathbf{a}}, \tilde{\mathbf{p}})$	Public Key
$v \in \{0,1\}^L$	Nonce
$M \in \{0,1\}^L$	Message to be encrypted

Output:

$(\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3)$	Ciphertext

1: $seed \leftarrow H(v \mid\mid M)$
2: PRNG-init($seed$)
3: $(\mathbf{c}_1, \mathbf{c}_2) \leftarrow CPA_{\mathrm{enc}}(\tilde{\mathbf{a}}, \tilde{\mathbf{p}}, v)$
4: $\mathbf{c}_3 \leftarrow G(v) \oplus m_{\mathrm{cca}}$
5: **return** $(\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3)$

Algorithm 3 $\mathrm{CCA}_{\mathrm{dec}}$

Input:

$\tilde{\mathbf{r}}_{\mathbf{2}}$	Private Key
$(\tilde{\mathbf{a}}, \tilde{\mathbf{p}})$	Public Key
$(\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3)$	Ciphertext

Output:

$M \in \{0,1\}^L$	Decrypted message

1: $v' \leftarrow CPA_{\mathrm{dec}}(\tilde{\mathbf{r}}_{\mathbf{2}}, \mathbf{c}_1, \mathbf{c}_2)$
2: $M \leftarrow G(v') \oplus \mathbf{c}_3$
3: $seed \leftarrow H(v' \mid\mid M)$
4: PRNG-init($seed$)
5: $(\mathbf{c}_1{}', \mathbf{c}_2{}') \leftarrow CPA_{\mathrm{enc}}(\tilde{\mathbf{a}}, \tilde{\mathbf{p}}, v')$
6: **if** $(\mathbf{c}_1{}', \mathbf{c}_2{}')$ is equal to $(\mathbf{c}_1, \mathbf{c}_2)$ **then**
7: **return** M
8: **else**
9: **return** $fail$
10: **end if**

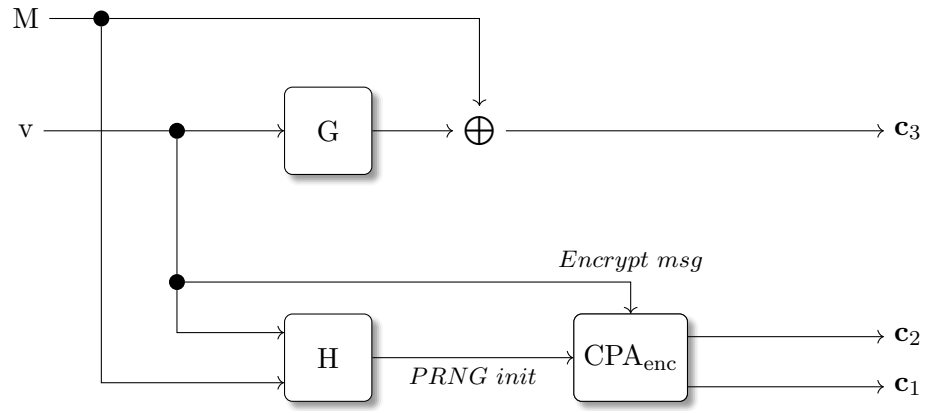


Figure 2.8: CCA2-secure encryption

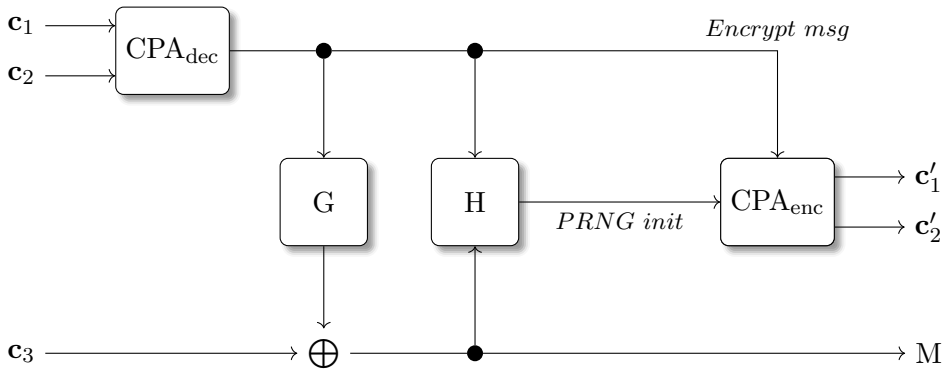


Figure 2.9: CCA2-secure decryption

2.8 Existing Hardware Implementations

This section provides an overview of existing hardware (FPGA) implementations of lattice-based encryption and signature schemes as well as hash functions.

In 2012, Györfi et al. presented a high-throughput hardware architecture for the SWIFFT / SWIFFTX ([45]) hash functions [32]. The presented a fully parallelized hardware implementation of an NTT focusing on high-throughput. However, this meant that the implementation was very large and required expensive FPGAs.

In 2012, Göttert et al. have presented the first complete hardware implementation of an LWE-based cryptosystem [29]. This work was a big step in assessing the practicality of lattice-based encryption. Their hardware implementation included multiplication in polynomial rings using a parallel implementation of the NTT. Using the NTT means a reduction of the runtime from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$. Their parallel implementation further reduced this to $\mathcal{O}(\log n)$. However, this meant that the implementation was very large and required expensive FPGAs. Furthermore, they explored different variants for sampling Gaussian distributed values (rejection sampling, a rounding-based approach, and a lookup table approach) in hardware and compared them to each other.

In the same year, Pöppelmann and Güneysu presented their work on an adaptable, extensible, and efficient NTT-based arithmetic for lattice-based cryptography [63]. Their design is much smaller and scalable. This served as a building block for their later work, in which they presented a fully functional efficient hardware implementation of Ring-LWE-based encryption [64]. Here, they showed that Ring-LWE encryption could be both cheap and fast in hardware. The design was based on a micro-code processor capable of the NTT-operation, addition, and subtraction of polynomials, as well as Gaussian sampling. It was designed as a versatile building block for future ideal lattice-based schemes. All parts of their implementation had a constant runtime, which protects against timing attacks. This was possible by implementing a constant-time Gaussian sampler using the inverse transform method. The same authors improved on this design further, by presenting a lightweight implementation, which was about ten times smaller [65].

In 2013, Aysu et al. presented area optimizations for the polynomial multiplication with the NTT [7]. Compared to previous work, they reduced the slice usage, the number of utilized memory blocks, and the number of memory accesses, by an improved memory organization and on-the-fly generation of operands.

Roy et al., in 2014, presented a compact coprocessor for a Ring-LWE-based encryption scheme [71]. They presented three optimizations to the NTT: avoiding preprocessing the input by merging the initial scaling operation with the main algorithm, reducing the fixed computation cost of the twiddle factors, and using an advanced memory access scheme using parallel FPGA-RAM slices to achieve maximum utilization of computational blocks. Additionally, they proposed optimizations to the Ring-LWE encryption scheme to reduce the number of NTT operations. Furthermore, they implemented a more efficient sampler using fast lookup tables, which is more compact and faster compared to previous work ([65]). Finally, targeting high-speed applications, they use pipelining to achieve a fast computation time.

In 2014, Pöppelmann et al. presented an FPGA implementation of Bimodal Lattice Signature Scheme (BLISS), a lattice-based signature scheme [66]. The authors presented techniques for efficient sampling of high-precision Gaussian noise. They integrated fast NTT-based polynomial multiplication, parallel sparse multiplication, and Huffman compression of signatures. As PRNG for their sampler, they chose Trivium. Keccak was used as a random oracle instantiation, due to its security and speed in hardware.

Lattice-based cryptography is one of the most promising quantum-secure candidates considered to be used as a replacement for currently used encryption schemes. Many different algorithms have been evaluated both in software and in hardware. However, ASIC designs were left unexplored. In 2016, Oder et al. discussed the many opportunities and challenges in implementing lattice-based cryptography on ASICs [51]. However, they did not present an implementation and many questions were left unanswered. For example, performance and area estimations were not given and it was unclear if such an implementation could achieve decent performance.

To answer these questions, this thesis presents a low-resource ASIC implementation of a lattice-based encryption scheme. The following chapter discussed the the requirements and basic design considerations for this implementation. Afterwards, the implementation details are presented, and the results are discussed. Concrete performance numbers are given both for the area and the timing.

Chapter 3

Requirements and Design Space Exploration

Compared to software and FPGA implementations, ASIC implementations have unique requirements that have to be met. These requirements, as well as fundamental design choices, are discussed in this chapter. First, in Section 3.1, the basic requirements an ASIC implementation has to fulfill are described, and the goals of this work are defined. Then, Section 3.2 will discuss some fundamental design choices. Afterwards, Section 3.3 will discuss different parameter sets for the instantiation of the encryption scheme and reason about the design choices made. Finally, in Section 3.4, an overview of the building blocks of the implementation is given.

3.1 Requirements and Goals

The goal of this thesis is to explore the practicality of Ring-LWE encryption schemes for constrained ASICs. For this purpose, the encryption algorithms presented in Section 2.5.2 and Section 2.7 are implemented as a coprocessor.

Possible real-world target applications of the coprocessor include Radio-frequency identification (RFID) chips and System on Chips (SoCs) for mobile devices. These target applications require both a small design and low power operation. ASICs are usually mass-produced, so it is critical to minimize circuit size to reduce manufacturing costs. Currently, there already exist some FPGA implementations of Ring-LWE encryption and signature schemes [7, 64, 65, 66, 71], however to our knowledge there exist no ASIC implementations of a Ring-LWE encryption scheme. A primary goal of this thesis is to enable a fair comparison with (highly optimized) ASIC implementations of other encryption schemes such as RSA and ECC.

The main focus of this thesis lies in minimizing the area while still providing acceptable performance for practical applications. To achieve these goals, it is important to choose efficient and small algorithms, and re-use modules for different tasks if possible. Additionally, as a secondary focus, the power consumption has to be kept as low as possible. For this purpose, clock-gating will be implemented. This reduces switching activity, which is the

main contributor to power consumption in Complementary Metal-Oxide-Semiconductor (CMOS) technology. To increase performance without increasing the complexity of the implementation significantly, some trade-offs in regards to the area requirements have to be made.

When implementing a cryptographic scheme, it is also necessary to provide security against side-channel attacks. This will be achieved by having all operations run in constant (data independent) time. However, other side-channel countermeasures, like masking [69, 52], were not considered to be within the scope of this thesis to reduce the complexity of this first ASIC implementation.

To summarize, the primary goals of this work is to have a fully-fledged Ring-LWE encryption coprocessor that is capable of encryption and decryption using two different Ring-LWE encryption algorithms. One of them is secure against adaptive chosen-ciphertext attacks, while the other only protects against chosen-plaintext attacks. Both encryption schemes will run in constant time. Algorithms will be chosen based on this constant-time requirement. The coprocessor will use the Advanced Microcontroller Bus Architecture (AMBA) [6] interface to communicate. Key-generation will not be a part of the design, as keys are often generated elsewhere.

3.2 Basic Design Considerations

This section will enumerate the required building blocks and algorithms, as well as describe basic design considerations for implementing them and explain why specific algorithms were chosen. Implementation details will follow in the next chapter.

For the NTT and the point-wise multiplication, as well as the addition of polynomials, a multiplier that can multiply numbers up to the selected prime q, as well as adder and subtraction units are required. Because all operations happen in a prime field, the modulo operation is also needed.

Beside the central arithmetic unit, several other modules are required: For generating error polynomials, a noise sampler is necessary. The entropy source for the noise sampler will be a PRNG. The PRNG will be integrated into the coprocessor because it needs to be constant time and secure against side-channel attacks as well. The CCA2-secure encryption scheme additionally requires a hash function. Lastly, memory and a control unit will also be part of the coprocessor.

The remainder of this section discusses the design choices and chosen algorithms for these modules.

Memory.
Memory in the form of RAM is required for the hashing algorithm, as well as for the Ring-LWE operations to store the keys and other polynomials. Compared to other encryption schemes, memory requirements for Ring-LWE-based encryption schemes are very high.

Current FPGA implementations typically use one or more instances of an on-chip dual-port block RAM. These implementations [63, 64, 65] are heavily optimized and rely on this dual-port block RAM. Since FPGAs already provide such blocks, this is a sensible option. However, for ASICs, many more options exist which allow tradeoffs regarding the area and performance.

For this work, due to the big memory requirements and the focus on a low area, a single instance of a single-port static random access memory (SRAM) macro was chosen for the main storage. SRAM macros typically use fewer resources than standard-cell based memories, so choosing a macro additionally reduces the area requirements. Dual-port SRAM macros exist and would allow for huge speed-ups and optimizations. However, they are about twice the size. This choice has a significant impact on the design of this coprocessor. A single-port RAM is limited to reading or writing a single word in each cycle, concurrent reads and writes are not possible. To avoid bottlenecks, it will be important to utilize the given RAM bandwidth as much as possible and minimize memory wait cycles. To reduce memory requirements, memory locations will be chosen carefully, and the Arithmetic logic unit (ALU) will share the memory with the hashing algorithm.

The exact size of the memory will be discussed later in this chapter, as it differs depending on the used parameter set.

Datapath and memory width.
For practical reasons, only memory widths of 8, 16, and 32 bits were considered for the memory width. This makes it easier to integrate the coprocessor into other blocks. For this work, 16-bit was chosen for the width of the RAM and the datapath because it is a very common width for microprocessors. Additionally, most proposed parameter sets for Ring-LWE encryption algorithms use primes that fit into 16 bits. This implementation will typically store one polynomial coefficient in a single RAM-word. While this limits the implementation to use primes no larger than 16 bits, it also significantly reduces complexity.

Multiplier.
The encryption scheme requires frequent multiplication of two $\log_2(q)$-bit words, where q is the chosen prime. This multiplier is the core of the ALU.

The hardware requirements of a simple $n \times n$ bit multiplier are roughly n^2 AND gates and adders. Despite the large area requirements, this work will use such a $\log_2(q) \times \log_2(q)$ bit integer multiplier. Using a half-width multiplier and using multiple half-width multiplications can also be a valid option for constrained devices. This would reduce the area requirements significantly. Furthermore, the single-port RAM is not fast enough to provide two $\log_2(q)$-bit operands in a single cycle regardless. However, using a full-sized multiplier reduces the complexity of the design. The multiplier is also heavily used for the chosen reduction algorithm. In the NTT operation, the multiplier will be used to calculate the so-called twiddle-factors. These can be calculated without fetching operands from the RAM. The reduction algorithm requires a double-width multiplication, which will be split up into multiple smaller ones, while all other multiplications are single-width. For this module, area was traded in for a speed-up in multiplications. Because not every multiplication requires fetching operands from RAM and modulo reductions are needed after every multiplication, the big multiplier can be justified easily.

It is important to make sure the multiplier gets used in as many cycles as possible to avoid any bottlenecks. For this purpose, the required operands will be preloaded into dedicated registers whenever possible. The multiplier is expected to be utilized for over 60% of the cycles during decryption.

Reduction algorithm.
For the modulo reduction, the Barrett reduction method was chosen for this design because it can re-use the multiplier in the ALU. Any other reduction algorithm, for example the Montgomery reduction, would have added more gates. Additionally, the Barrett reduction can be easily implemented in constant time, which helps strengthen the implementation against side-channel attacks. When designing for a single parameter set, an algorithm specifically tailored to one prime could be used. However, due to the lack of a standardized parameter set, it is currently advantageous to use a generic algorithm that supports many different parameter sets. The specific Barrett reduction algorithm used in this work is shown in Algorithm 4.

Algorithm 4 Barrett Reduction

Input:
a Integer to be reduced. $a \leq q^2$
Output:
r $a \bmod q$

1: $x \leftarrow (a \cdot \mu) \gg k$
2: $r \leftarrow a - (x \cdot q)$
3: **if** $q \leq a$ **then**
4: $\quad r \leftarrow r - q$
5: **end if**
6: **return** r

Here, μ is a precomputed constant which is defined as $\mu = \lfloor 2^k/q \rfloor$, where q is the modulus. The Barrett reduction uses the constant μ to approximate a division of a/q. Due to this approximate nature, the algorithm can only guarantee a correct result for a specific range of inputs. The *error* of the Barrett reduction can be calculated by $1/q - \mu/2^k$. The maximum value for a valid input a is thus given by $\lfloor 1/error \rfloor$. The value k has to be chosen, such that $\lfloor 1/error \rfloor \geq q^2$.

In this work, k is chosen to be $2\ \log_2(q)$. By using this value for k, μ will be exactly $1 + \log_2(q)$-bit wide. Line 2 in this algorithm is a double-word multiplication. Splitting up $a \cdot \mu$ into two smaller single-word multiplications results in multiplications of up to $\log_2(q) \times (\log_2(q) + 1)$-bit. This results in a 1-bit increase of the width for one operand of the multiplier to accommodate the reduction algorithm. Therefore, in terms of area, this reduction algorithm comes at a very low cost.

Noise sampler.
The sampler is required for generating error polynomials, which are used during encryption. For this purpose, a binomial sampler is used. Binomial samplers are very simple and efficient, have no data-dependent branches and run in constant time. Due to these benefits binomial samplers are used in many efficient lattice-based designs [4, 52]. Samples from a binomial distribution ψ_k are calculated by computing $\sum_{i=0}^{k-1} b_i - b'_i$ where $b_i, b'_i \in \{0, 1\}$ are $2k$ uniform independent bits provided by a PRNG. For a standard deviation of $\varsigma = \sqrt{k/2}$, where $k \in \mathbb{Z}$, this algorithm requires $2k$ bits from an entropy source.

Hash function.
A hash function is required by the CCA2-secure encryption algorithm described in Section 2.7. For this work, Keccak was chosen in the SHAKE128 instantiation. Keccak is the winner of the SHA-3 competition by the National Institute of Standards and Technology (NIST). Many lattice-based encryption and signature implementations [11, 52, 4] use Keccak as a hash function due to its excellent performance, security properties, and flexibility.

The specific implementation [60] used in this work was authored by Peßl and Hutter and was adapted for use as an extendable output function (SHAKE). It was chosen because it is a very small and efficient hardware implementation of the Keccak algorithm, which supports the chosen memory bus width of 16-bit. This implementation of SHAKE128 requires 1600 bits (200 Bytes) of memory, which means it requires 100 words of the RAM. Due to the extensive memory requirements, it was decided to share the RAM between the ALU and the hash function. The memory layout and order of operations are designed in such a way to make sure there is always free memory for the hash function, while still minimizing the memory requirements for the coprocessor as a whole. During hashing operations, the hash function will have exclusive access to the RAM.

PRNG.
A PRNG is required as an entropy source for the noise sampler. PRNGs can be based on block and stream ciphers, hash functions, or dedicated algorithms.

Reusing Keccak as a PRNG is not an option due to performance reasons. Using Keccak would result in roughly 95% of the runtime of the encryption to be spent on generating pseudo-random values for the sampler. Instead, this work uses Trivium as a PRNG. Trivium is a very lightweight stream cipher with a flexible trade-off between area and speed. It is was already used as a PRNG in other Ring-LWE implementations (e.g. [65, 66]).

The chosen Trivium implementation can generate between 2^0 and 2^6 bits per cycle while having a size of just 3–5kGE. While this work allows Trivium to be instantiated with any bits per cycle within that range, for the purpose of performance and area evaluations, 16 bits per cycle were chosen for this work. This offers excellent performance, and the additionally required area is insignificant compared to other modules of the coprocessor. Compared to this choice, instantiating it with 1 bit per cycle would save an area of up to 1kGE. In practice, it has to be carefully evaluated if the trade-off is worthwhile, based on the design goal and the application.

In this work, the implementation was taken from [80] and adapted to allow for re-seeding to be split up into multiple cycles. This change was made to avoid having to store the entire 160 bits that are needed for seeding the PRNG in registers.

3.3 Parameter Selection

Ring-LWE encryption is still relatively new, and to this date, no standardized algorithm and parameter set exist. However, many different parameter sets have been proposed [48, 41, 29, 75, 76, 4, 52]. This section will briefly explain what a parameter set is, which sets have been previously proposed, and which parameter sets will be used in this work.

The Ring-LWE encryption schemes presented in Section 2.5.2 and Section 2.7 are parameterized by the three variables σ, n, and q. These three parameters together represent the so-called parameter set and define the security level of many Ring-LWE encryption schemes. To be exact, the security level (hardness of the Ring-LWE problem) depends on the dimension n, as well as the ratio of q/σ, where a bigger n or a smaller ratio means an increase in security.

In literature, the parameter s is often given instead of σ. It is calculated by $s = \sigma \cdot \sqrt{2\pi}$. Conversely, σ is calculated by $s/\sqrt{2\pi}$.

We recall from Section 2.6 that the standard deviation of binomial distributions are denoted as ς, whereas Gaussian distributions use σ. For Ring-LWE, when comparing security levels of different distributions that roughly follow a discrete Gaussian, the standard deviation (as denoted by σ and ς) is the most important factor [52]. While they are not exactly equivalent, in this work σ values from proposed parameter sets that use a Gaussian distribution are approximated to the closest ς that the sampler used in this work allows. This allows for a comparison of many more different parameter sets.

Authors	n	q	σ	ς	k	Public Key Size	Security Level	Set
Lindner and Peikert[41]	128	2 053	2.70			1 536 bits	$\ll 128$ bit	
	192	4 093	3.53			2 304 bits	< 128 bit	
	256	4 093	3.33			3 072 bits	≈ 128 bit	
	320	4 093	3.19			3 840 bits	> 128 bit	
Göttert et al.[29]	256	7 681	4.51	(4.52)	41	3 328 bits	≈ 128 bit	P_1
	512	12 289	4.85	(4.85)	47	7 168 bits	≈ 256 bit	P_2
Micciancio and Regev[48]	136	2 003	5.19			1 496 bits	≥ 128 bit	
	214	16 381	2.94			2 996 bits	≥ 128 bit	
Singh [75, 76]	512	25 601	3.19	(3.24)	21	7 680 bits	≈ 128 bit	P_3
	1 024	40 961	3.19	(3.24)	21	16 384 bits	≈ 256 bit	P_4
Oder et al.[52]	1 024	12 289		2	8	28 672 bits	≈ 256 bit	P_5
Alkim et al.[4]	512	12 289		3.46	24	7 168 bits	≥ 128 bit	P_6
	1 024	12 289		2.82	16	14 336 bits	≥ 256 bit	P_7

Table 3.1: Various parameterization proposals for Ring-LWE-based encryption schemes. The parenthesized values in the ς column are calculated by $\varsigma = \sqrt{k/2}$, where $k \in \mathbb{Z}$ is the value used for the binomial sampler.

Table 3.1 lists many previously proposed parameter sets offering varying security levels. The estimated security levels of these parameter sets are taken from the security analysis from [41, 79] or directly from the authors of the proposed sets themselves.

Since there exists no standardized parameter set yet, this work aims to support as many different parameter sets as possible, to allow for a fair comparison. The sets that are compatible and can be used with this implementation are named P_1 through P_7 in the rightmost column of Table 3.1.

The parenthesized values in the ς column are calculated as $\varsigma = \sqrt{k/2}$, where $k \in \mathbb{Z}$ is the value used for the binomial sampler. It is chosen such that the difference between a binomial distribution with a standard deviation of ς and a Gaussian distribution with a standard deviation of a given σ is minimal.

Some parameter sets cannot be tested due to having a dimension n that is not a power of two, or because $q \equiv 1 \mod 2n$ does not hold. In that case, no n-th root of unity exists, and the NTT cannot be used. Additionally, the chosen bus width excludes schemes that use primes with more than 16 bits. Using fixed primes would allow for optimized arithmetic and specialized algorithms, however, to allow for many different parameter sets to be compared, generic algorithms were chosen wherever required. Supporting many different parameter sets at varying security levels allows for comparison with state of the art implementations and different encryption schemes like RSA and ECC. The results of this comparison will be presented in Chapter 5.

3.4 Implementation Overview

To conclude this chapter, this section will summarize the most important facts about the implementation and give a high-level overview of its structure.

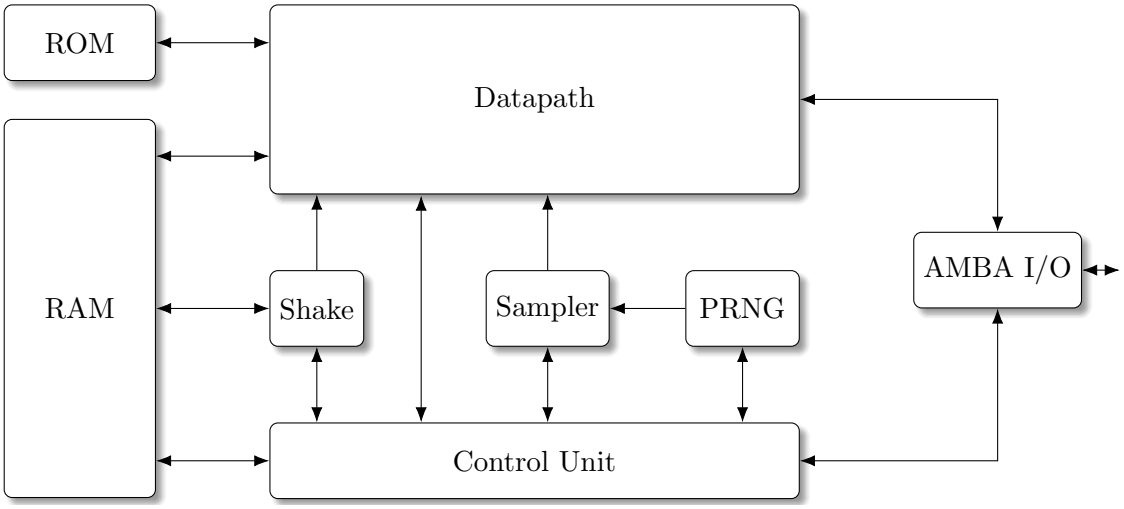


Figure 3.1: A high-level overview of the implementation.

Figure 3.1 shows the overall structure of the implementation. It is split into the following major modules: Datapath, Read-only Memory (ROM), RAM, the hashing function, sampler, PRNG, control unit and the AMBA Advanced Peripheral Bus (APB) interface.

The datapath houses a few small registers and the ALU, which contains logic required for multiplication, addition, subtraction and modulo reduction. The previously unmentioned ROM contains a small $\log_2(q) \cdot \log_2(n)$-bit sized lookup table, which stores precomputed values to speed up the NTT operation. Beside a few registers within the datapath itself, the RAM is used as the main memory and has a size of $130 \cdot n$-bit. It is shared between the hashing function and the ALU. Only one of these two modules has exclusive access to the RAM at any given time.

For the hashing function, Keccak is chosen in the SHAKE128 variant. The specific implementation by Peßl and Hutter [60] is taken and integrated into the coprocessor. For the sampler, which is used to generate error polynomials, a binomial distribution is used as an approximation to a Gaussian distribution. The PRNG function is provided by the Trivium stream-cipher. Its sole purpose is to provide random numbers for the sampler. The

Trivium implementation from van Rantwijk [80] is adapted and integrated. The AMBA bus is a simple, standardized interface implemented according to its specification [6]. The bus has an interface width of 16-bit, which is the same width as the RAM and the datapath. The bus is used for controlling the coprocessor, querying the status, as well as for writing and receiving messages. The control logic is implemented as a number of finite-state machines handling individual sub-tasks. It also controlls access to the RAM to switch between the hashing module and the ALU.

The coprocessor implements the Ring-LWE encryption scheme defined in Section 2.5.2. It can be instantiated with various parameter sets, and with as well as without support for the CCA2-secure scheme described in Section 2.7. In the second case, the hashing module will be removed, and the RAM will be sized differently.

Given this overview, the next chapter will describe the implementation details for each of the presented modules and explain the sub-tasks required to enable encryption and decryption.

Chapter 4

Implementation Details

After introducing the basics of lattice-based cryptography in Chapter 2, and defining the basic requirements of the implementation as well as explaining fundamental design choices in Chapter 3, this chapter will dive into the implementation details.

First, details about the datapath and the ALU are given in Section 4.1. Section 4.2 explains the implementation details about the reduction algorithm and how the ALU is used. Then, in Section 4.3, the implemented NTT algorithm, including all its optimizations, is presented. Section 4.4 describes the memory layout and how elements are stored in the RAM. Afterwards, Section 4.5 explains in detail how the two pre-existing modules Keccak and Trivium were integrated and how they are used. Section 4.6 shows how the noise sampler generates its values using the PRNG. Finally, the control unit is presented in Section 4.7. Here, the Finite-state Machines (FSMs) that are used for encryption and decryption are explained in detail. Additionally, the exact memory layout during each operation is presented.

4.1 The Datapath and the Arithmetic Logic Unit

This section will explain the details of the datapath. It will explain the connections to other modules and which functionalities it provides for the coprocessor.

The datapath is the core of the coprocessor and consists of some registers and the ALU. As already shown in Figure 3.1, the datapath is connected to the control unit, sampler, hashing-function, RAM, ROM, and the AMBA-interface. The remainder of this section will go into details about the contents of the datapath, as well as explain the functionality it provides, and which mathematical operations are supported.

Arithmetic Logic Unit.
The ALU provides basic operations like multiplication, addition, subtraction, XOR, as well as RLWE-encoding and decoding. It is controlled by instructions sent from the control unit. Based on the instruction, inputs for all arithmetic operations, as well as the inputs for the registers and the RAM, are selected. The instructions are designed, such that the number of possible inputs for each operation, and thus the size of the multiplexers, is at a minimum.

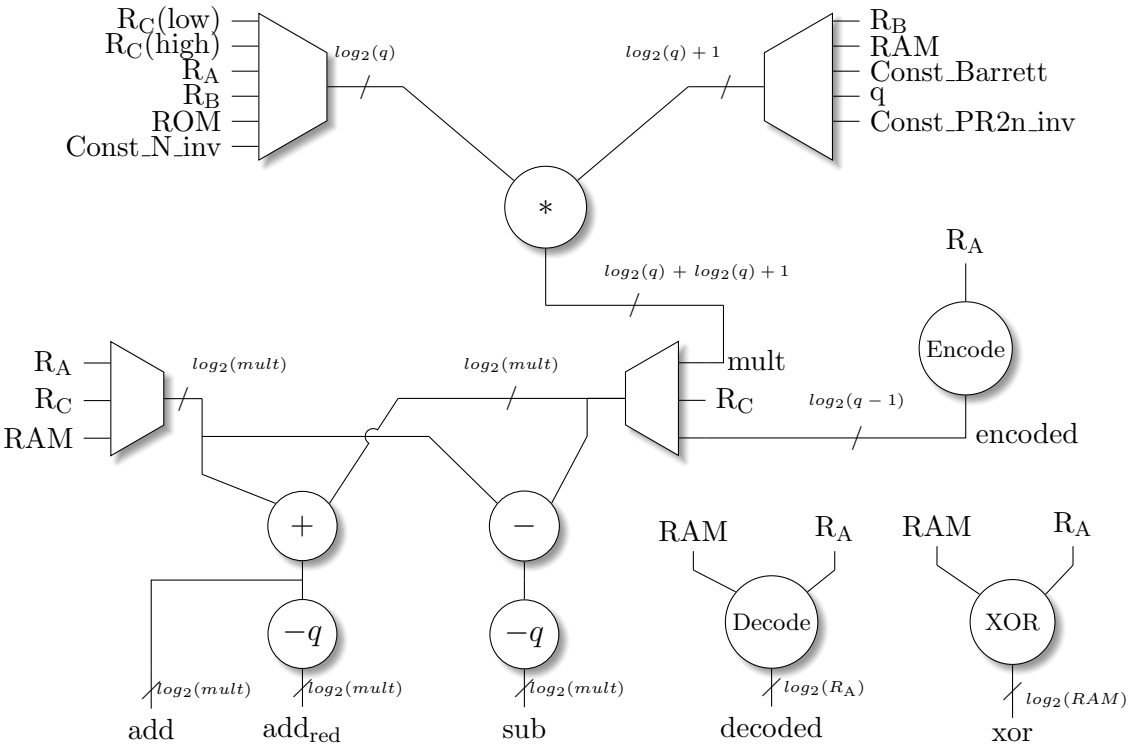


Figure 4.1: A graphical representation of the ALU

The ALU was designed to be optimized for the NTT butterfly operation and the reduction algorithm. Both require multiplications followed by a subtraction or addition. For this reason, the ALU was constructed in a way, such that a multiplication, followed by an addition or subtraction, is possible within a single cycle.

A graphical representation of the ALU is shown in Figure 4.1. For simplicity, the registers are not shown. In this figure, Const_Barrett represents the constant μ from the Barrett reduction algorithm described in Section 3.2. Further constants are the prime q, the $2n$-th root of unity Const_PR2n_inv, and the modular multiplicative inverse of n Const_N_inv. Registers are denoted as R_A, R_B, and R_C.

Registers.
The datapath uses three registers (R_A, R_B, R_C) for storing results and operands. R_A has a size of 16 or $\log_2(q) + 1$ bits, depending on which is bigger. R_B is $\log_2(q)$ bits wide, and R_C is $2\log_2(q)$ bits wide. The registers were sized as small as possible, and each register serves a specific purpose, which changes depending on the instruction. For instance, R_C is is much bigger than the other two because it is used to store multiplication results. The registers can store values from the RAM, AMBA-interface, hash function, or results from the ALU.

Addition and Subtraction.
Inputs for the first operand for the addition and subtraction are R_C, R_A, or the current RAM output. The second operand takes its input from R_C, the result of the multiplication or encoder function. Both results are optionally modulo reduced by 1 bit. This happens by a multiplexer that chooses the reduced result (result$-q$) if the sum is greater than q. In

software, this would not be ideal because it could leak timing information, but in hardware, it can be done in constant time. Having parallel addition and subtraction and the duplicate reduction logic greatly improves performance for the NTT operation and simplifies the instruction set of the ALU. In order to decrease the size of the ALU, both the subtraction and addition use the same inputs. This reduces the total number of multiplexers significantly.

Encode and Decode.
The Ring-LWE-encode and decode functions, as described in Section 2.4.3, are implemented as part of the ALU. Messages to be encoded are stored in R_A. Depending on the current bit position, the bit-value is mapped to $q/2$ if it was 1, or 0 otherwise. The result is then used as an input for the addition/subtraction. For decoding, a value from the RAM is decoded into a single bit, which is appended to the right of the value from R_A. Results are written back to R_A or the RAM.

Multiplication.
The multiplication functionality is provided by a $\log_2(q) \times (\log_2(q) + 1)$-bit multiplier. Its inputs are taken from the registers, the RAM, or constants. In the case of R_C, either the lowest or highest $\log_2(q)$ bits are taken. The result of the multiplication is either stored in R_C or directly used as an input for the addition/subtraction. Addition/Subtraction after multiplication is a very common instruction in the NTT operation and the chosen reduction algorithm, which is why they are computed in the same cycle.

Placing a register between these steps could reduce the critical path and thus increase the maximum clock speed. However, this would also mean adding an additional $2\log_2(q)$-bit register which would increase the total number of register bits by 50%. Additionally, due to the chosen reduction algorithm, this would also have a negative impact on the total runtime despite the increase in the clock speed, as the NTT operation would run about 30% slower. For these reasons, it was not implemented.

4.2 Modular Reduction

As described in Section 3.2, the Barrett reduction algorithm is used for the reduction after multiplication. The multiplication $R_C \cdot \mu$ in Algorithm 4 is split up into two smaller multiplications to fit the multiplier of the ALU. Values to be reduced are always stored in the register R_C, and the result is placed in R_A. The Barrett reduction algorithm uses three ALU instructions and completes in three cycles, without needing any additional logic. The three instructions are shown in Algorithm 5.

For a better understanding of how the multiplication is split up, Figure 4.2 gives a detailed visualization of the procedure. We recall that R_C is $2\log_2(q)$ bits wide, which is equal to the constant k and the width of the multiplier output. In this figure, $R_{C_{high}}$ represent the upper k bits of R_C and $R_{C_{low}}$ the lower k bits of R_C. Both of these halves are multiplied with μ separately, to fit the width of the multiplier, and then added together. The upper k bits of the result of the addition are then equal to $(R_C \cdot \mu) \gg k$. This result is stored in the register R_A and then used for line 3 in Algorithm 5. Note, that the original

algorithm (Algorithm 4) requires an additional modular reduction after the last subtraction. However, for this implementation, all subtraction results are automatically reduced as already shown in Figure 4.1.

A modular reduction is required after almost all multiplications, so it needs to be fast, and it is important to utilize as many resources as possible during its runtime and use pipelining where possible. This implementation utilizes the multiplier of the ALU in 100% of the cycles and uses two out of the three registers. During this operation, the third register (R_B) is usually also utilized by filling it with a value from the RAM, which is needed in subsequent instructions.

Algorithm 5 Barrett reduction algorithm in detail

Input:

R_C	Register that contains the integer to be reduced.
q	Modulus
k	Constant $2\log_2(q)$
μ	Constant $\lfloor 2^k/q \rfloor$

Output:

R_A	$R_C \bmod q$

1: $R_A \leftarrow (R_{C_{low}} \cdot \mu) \gg k/2$
2: $R_A \leftarrow ((R_{C_{high}} \cdot \mu) + R_A) \gg k/2$
3: $R_A \leftarrow R_C - (R_A \cdot q)$

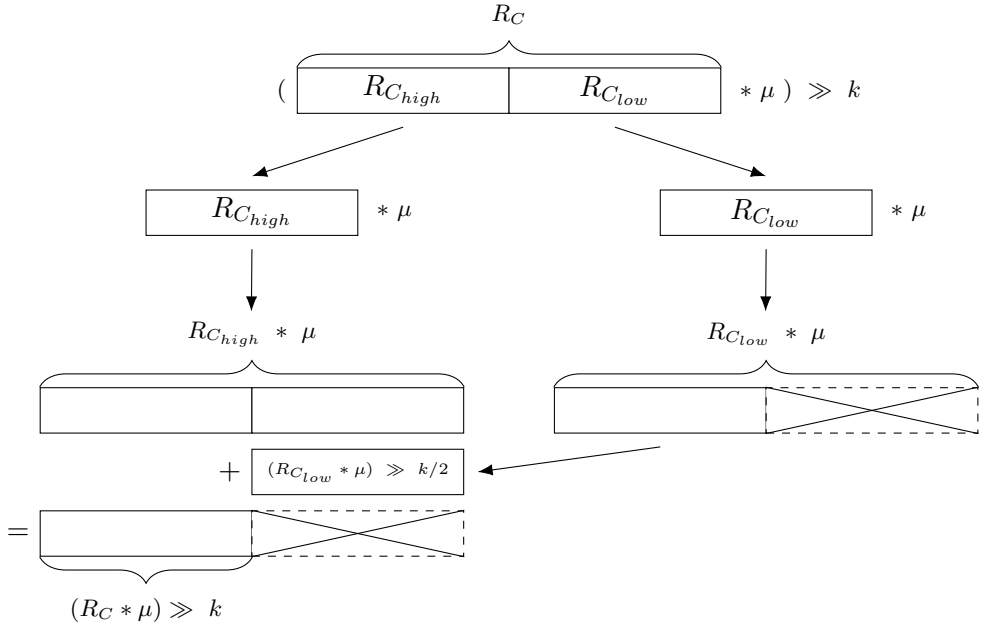


Figure 4.2: A visual representation of the split-up multiplication $(R_C * \mu) \gg k$ in the reduction algorithm

4.3 NTT

Compared to the NTT algorithm presented in Section 2.5.1, three optimizations have been applied to the algorithm in this work. First, the initial scaling step for the forward-NTT has been eliminated by integrating it with the NTT. Second, ω_m values from line 3 in Algorithm 1 have been precomputed to reduce the number of multiplications. Also, the *BitReverse* step is only applied when necessary. When the input to the NTT is an error polynomial, then the *BitReverse* step can be omitted because changing the order of the polynomial coefficients makes no practical difference.

Precomputing $\omega_\mathbf{m}$.
In Algorithm 1, ω_m has to be computed in every layer of the butterfly network, this happens a total of $n-1$ times per NTT operation and equates to roughly 15% of the multiplications within the NTT.

By precomputing these values and storing them in a lookup table, these multiplications can be eliminated by trading it with a $2 \cdot \log_2(q) \cdot \log_2(n)$ bit sized lookup table. When increasing the lattice dimension n, this will only grow at a rate of $\mathcal{O}(\log n)$. Similar to the approaches in [44, 63], in this work, these precomputed values are stored in a ROM.

While the $\omega = \omega \cdot \omega_m$ calculation could also be removed by precomputing all possible twiddle factors, this would take up a much larger area for the ROM. When also considering the constants required for scaling (see below), then a $2n \cdot \log_2(q)$-bit sized ROM is needed to store all these values. Compared to the previous optimization, for a lattice dimension n of 256, the ROM would be 31 times bigger, and for dimension n of 1024, the ROM would grow by a factor of 102.

In this work, due to the increased area, this option was not pursued, even though this calculation equates to 16–19% of the total multiplications of the NTT. However, for a performance-oriented implementation, precomputing all values for ω makes sense, as it would significantly cut down on the total runtime for encryption and decryption.

Scaling.
Instead of first computing the scaled polynomial and then performing a standard NTT, Roy et al. proposed to integrate this step directly into the NTT operation [71]. This can be done by simply initializing ω in line 4 of Algorithm 1 with ω_{2m}, the $2n$-th primitive root of unity, instead of the value 1. It must be noted that this only works for the forward-NTT. The INTT still requires an additional scaling step.

Final Algorithm.
The final NTT algorithm, which includes all these optimizations, is shown in Algorithm 6. In line 17, ROM(mode, $\log_2$(m)) represents a table-lookup from the ROM. Line 20 to 26 show the additional scaling step needed for the inverse-NTT.

In this work, the NTT operation is implemented as an FSM within the control unit. The *BitReverse* operation, as well as the *Scaling* for the inverse NTT, are implemented as sub-FSMs which get invoked by the main NTT-FSM.

Using the reduction algorithm presented in Section 4.2 and the NTT algorithm given in Algorithm 6, a single butterfly operation takes 6 cycles. This includes reading coefficients from the RAM, multiplying one coefficient with ω and reducing the result modulo q, adding and subtracting the previous result to the other coefficient, and writing the two resulting coefficients back to the RAM.

Algorithm 6 Optimized version of an iterative $n-$coefficient NTT

Input:

$\mathbf{x}$	Polynomial $\in \mathbb{Z}_q^n$
ω_n	n-th primitive root of unity $\in \mathbb{Z}_q$
ω_{2n}^{-1}	$2n$-th inverse primitive root of unity $\in \mathbb{Z}_q$
n^{-1}	Modular multiplicative inverse of n
$isErrorPolynomial$	Boolean value stating if the input $\mathbf{x}$ is an error polynomial
$mode$	Mode of operation ($forward$ or $inverse$)

Output:

$\widetilde{\mathbf{x}}$	Polynomial $\in \mathbb{Z}_q^n = \text{NTT}(\mathbf{x})$

if not $isErrorPolynomial$ **then**
  $\widetilde{\mathbf{x}} \leftarrow \text{BitReverse}(\mathbf{x})$
end if

for $m = 2$ to n by $m = 2m$ **do**
  if $mode = forward$ **then**
    $\omega \leftarrow 2n$-th primitive root of unity
  else
    $\omega \leftarrow 1$
  end if
  for $j = 0$ to $\frac{m}{2} - 1$ **do**
    for $k = 0$ to $n - 1$ by m **do**
      $t \leftarrow \omega \cdot \widetilde{\mathbf{x}}[k + j + \frac{m}{2}]$
      $u \leftarrow \widetilde{\mathbf{x}}[k + j]$
      $\widetilde{\mathbf{x}}[k + j] \leftarrow u + t$
      $\widetilde{\mathbf{x}}[k + j + \frac{m}{2}] \leftarrow u - t$
    end for
    $\omega \leftarrow \omega \cdot ROM(mode, \log_2(m))$
  end for
end for

if $mode = inverse$ **then**
  $s \leftarrow n^{-1}$
  for $i = 0$ to n **do**
    $\widetilde{\mathbf{x}}[i] \leftarrow \widetilde{\mathbf{x}}[i] \cdot s$
    $s \leftarrow s \cdot \omega_{2n}^{-1}$
  end for
end if

4.4 Memory Organization

This section describes the memory layout of the RAM.

The implemented encryption schemes have a message size of n bits, equal to the lattice dimension n. The RAM is used to store these messages, as well as all polynomials. These elements either have a size of n words, in the case of polynomials, or n bits for messages. Polynomials with n coefficients are stored in n words of the RAM, where each coefficient has the same size as the word size of the RAM, i.e., 16 bits. This limits the possible parameter sets to be used with this implementation to primes of up to 16 bits. However, most practical parameter sets fit into this category.

For the CPA-secure encryption scheme, a total memory of $97n$ bits, or $6n + \frac{n}{16}$ 16-bit words, is required. This consists of three polynomials for the public and private keys, another three polynomials for storing $\mathbf{c}_1$, $\mathbf{c}_2$, and $\mathbf{e}_1$, as well as a $\frac{n}{16}$ word-sized portion to store the n-bit message.

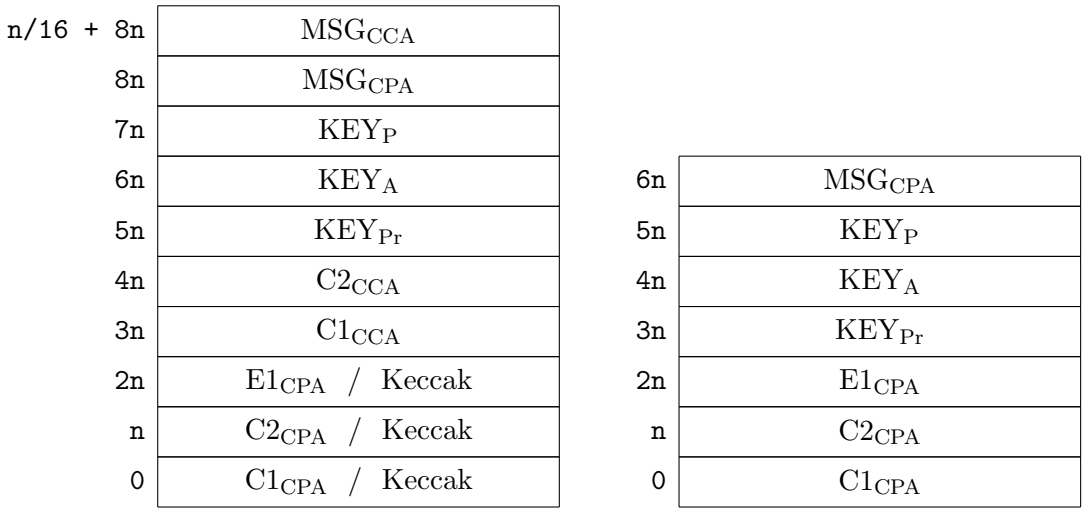
When instantiating the implementation with support for the CCA2-secure encryption scheme, an additional two polynomials with n words each and an area of $\frac{n}{16}$ words is required. The additional memory is used for storing the inputs $\mathbf{c}_1$ and $\mathbf{c}_2$, as well as the additional arguments $\mathbf{c}_3$ and v. In total, $130n$ bits, or $8n + 2\frac{n}{16}$ 16-bit words, are required.

The public key ($\mathrm{KEY_P}$, $\mathrm{KEY_A}$) and the private key ($\mathrm{KEY_{Pr}}$) are stored in the RAM instead of the ROM because keys are typically generated elsewhere. This also offers greater flexibility, as the keys can be changed very quickly.

Memory layout.
The RAM is organized in different regions of n or $n/16$ words. Polynomials, which have a size of n words each, are stored beginning at adress 0. The message, and in the case of enabled CCA2-security also the nonce, have a size of n-bit each. These are stored the the upper adresses directly after the polynomials. The exact memory layout is depicted in Figure 4.3. For intantiations without CCA2-security, this means that the lowest $6n$ words are used to store polynomials, while the message is stored beginning at adress $6n$. In the case of added CCA2-security, the lowest $8n$ words are used to store polynomials. Beginning at address $8n$, two n-bit sized regions are reserved for the message and the nonce v. Each of the regions is named to clearly differentiate between the different memory areas, the chosen names represent which data is typically stored at their locations, but the actually stored data depends on the respective operation. Because exact addresses depend on the lattice dimension n of the chosen parameter set, the rest of this thesis will refer to these region names rather than exact addresses.

Because the lattice dimension n is always a power of two, all polynomials are aligned in memory and accesses to coefficients require no addition. The address of the polynomial can simply be concatenated with the index of the coefficient.

For the CCA2 encryption/decryption, memory locations are chosen in a way, such that the hashing function, Keccak, can always use the lowest 1600 bits of the RAM. This way, no additional memory-address adder is required. As Figure 4.3 shows, Keccak uses the same memory area as the first three polynomials. This works for parameter sets where the lattice dimension n is at least 64. When using smaller dimensions, an additional buffer is introduced between the 3rd and 4th polynomial, such that the 1600-bit that Keccak needs, do not overlap with the 4th polynomial. This buffer is sized such that the 4th polynomial is located at a memory address aligned to n words. While dimensions of less than 256 are not practical, this allows smaller dimensions to be evaluated as well.



(a) Memory layout with CCA2

(b) Memory layout without CCA2

Figure 4.3: Memory layout of the RAM. The numbers on the left are the addresses, n is the lattice dimension from the chosen parameter set. The total memory size is either $8n + 2n/16$, or $6n + n/16$ words.

4.5 Integration of Keccak and Trivium

This section explains in detail how the pre-existing modules Keccak and Trivium were integrated into this implementation.

As discussed in Chapter 3, Keccak is used as the hashing algorithm. Trivium is a stream cipher used as the PRNG, which is then used to generate binomially distributed samples with the sampler. Integrating external blocks is not always trivial, because many blocks do not use any standardized interfaces. These two blocks have to interact with other modules such as the ALU, RAM, and the sampler. The control unit is responsible for managing the control signals for these two modules, as well as for switching the RAM access between Keccak and the ALU.

4.5.1 Keccak

The chosen Keccak implementation, authored by Peßl and Hutter [60], was integrated into this work for use as a hashing function used by the CCA2-secure encryption algorithm. Keccak is connected to the control unit, the datapath, and the RAM. The control unit is responsible for driving the control signals of the Keccak module, as well as for switching the RAM-access between the ALU and the Keccak module.

The memory layout (Figure 4.3) shows that Keccak uses the lowest $3n$ words of the RAM. However, this area is not exclusive to the Keccak module. During encryption and decryption operations, this memory area is used to store other values before and after the hashing operation. To keep the RAM size at a minimum, memory addresses had to be chosen carefully, and operations had to be ordered in a way, such that during a hashing

operation, Keccak would not overwrite any used memory areas. Because the Keccak memory area begins at address 0, no additional memory address adder was required. The exact contents for each memory areas during encryption and decryption will be shown later in this chapter.

Keccak is used for hashing values from the RAM, for re-seeding the PRNG, and for using the hash output for other computations. The register R_A of the datapath is used for storing the output, as well as for providing the input of the hashing function.

The specific implementation used in this work has an 8-bit wide interface for the input and output, which means that reading or writing a single RAM-word requires multiple cycles. To hash values from the RAM, they are first loaded into the register R_A. The content of R_A is then used as the input for the hashing module. Because the input of the hashing module is only 8 bits wide, whereas the register can fit entire words of 16 bits, a different half of the register is taken for each byte. For the hashing output, the reverse happens. Each byte is appended to the value of R_A shifted by 8 bits. After reading or writing a complete word from the hashing module, the RAM access is switched back to the ALU, such that the result can be used for other operations.

For the CCA2-secure encryption scheme, the computation of $\mathrm{G}(v)$ and $\mathrm{H}(v||M)$ is required, where v and M are n bits each. The scheme further requires G and H to be different oracles. For this purpose, the first word that is sent to the hashing function is different for these different invocations. The output for $\mathrm{G}(v)$ is the same size as the message M, which is n bit. The result from the $\mathrm{H}(v||M)$ operation is used to re-seed the PRNG. For this purpose, 160 bits are taken from the hashing module.

These operations are implemented as separate FSMs within the control unit. Depending on the state of the FSM, the RAM access is also switched between the ALU and the Hashing module, such that the hash function has access to the RAM when needed, and the ALU has access when reading values to be hashed or writing the results back to the RAM. The FSMs read and write to the control signals of the hashing module. These control signals show if the input or output is valid, if the hashing module is ready to take inputs, to acknowledge that the output has been taken, to signal the end of the input, and to reset the hashing module.

SHAKE128 uses 24 rounds of permutations, a rate (= block size) of 1344 bits, and a capacity of 256 bits. The used implementation requires around 15k cycles for these 24 rounds. For parameter sets up to $n = 512$, all hashing operations will take 15k cycles, because all hashing inputs fit inside 1344 bits. For parameter sets where n is bigger than 512, the $\mathrm{H}(v||M)$ operation will take twice that time, because v and M are both at least 1024 bits, which no longer fit the block size. The exact cycle counts for each operation and mode of operation will be presented in Chapter 5.

4.5.2 Trivium

The stream cipher Trivium is used as a PRNG, which acts as the entropy source for the noise sampler. While the used Trivium implementation is flexible and can generate between 2^0 and 2^6 bits per cycle, for this work, the module was instantiated such that it generates 16 bits per cycle. Trivium requires to be seeded with an 80-bit key and an 80-bit initialization vector.

The implementation by van Rantwijk [80] was modified to be re-seedable in multiple cycles to avoid a large 160-bit register. 160 bits are needed to re-seed Trivium. These are taken either from the user via the AMBA bus or directly from the output of the hashing function. To fit the bus-width of the rest of implementation, a 16-bit wide input was added to the module, to allow partial re-seeding. This input is directly connected to the general purpose register R_A. The re-seeding process is initiated and controlled by two FSMs in the control unit, one for each type of input (hash function or user input). After re-seeding, the module has to run for 72 ($= 4 \cdot$ size of the state / bits per cycle) cycles, before a valid output is generated. After that, it can generate 16 pseudo-random bits per cycle.

4.6 Sampler

The noise sampler creates samples from a binomial distribution ψ_k. Each sample is calculated by $\sum_{i=0}^{k-1} b_i - b'_i$ where $b_i, b'_i \in \{0, 1\}$ are 2 uniform independent bits provided by the PRNG. For a standard deviation of $\varsigma = \sqrt{k/2}$, $2k$ random bits are required per sample.

The sampler can take as many bits each cycle as the PRNG can provide. Remember that the PRNG generates 16 bits per cycle. The sampler needs $\lceil 2k/16 \rceil$ cycles to generate one sample. In each cycle, if the PRNG output is valid, 16 bits are taken from the PRNG. These bits are then subtracted from each other pairwise. The result is then added to an internal register. In the last iteration, if k is not a multiple of 2, only $2k$ mod 16 bits are used from the PRNG. Additionally, in the last cycle, the the prime q is added to intermediate result if it is smaller than 0. This ensures that the result is a valid value.

The sampler is directly connected to the PRNG, takes its output when required and when the PRNG output is valid, and then instructs the PRNG to generate the next value. Apart from the clock and reset signals, the module has one input to instruct it to generate a new sample, one output for the current sample, and another output signal to indicate that the sample output is valid.

The control unit has a separate FSM to control this module to generate binomially-distributed pseudo-random polynomials, which are stored in the RAM. When instructed, the FSM simply takes the values from the sampler and stores them in the RAM at a given address. After n samples, the random polynomial is complete, and the FSM returns to its idle-state.

4.7 Controller

The control unit is responsible for controlling each of the sub-modules and for the communication with the outside world through the AMBA bus. The provided functionality is implemented through a number of FSMs. The basic operations for encryption and decryption, both the CPA-secure and the CCA2-secure variants, have their own FSM. Operations are split up into smaller tasks, each with their own FSM. The four main FSMs for encryption and decryption will invoke these smaller FSMs when needed. Besides the encryption algorithms, the main controller provides functionality for loading values from the RAM, storing values in the RAM, as well as for re-seeding of the PRNG. The main controller is a separate FSM, which invokes the other FSMs depending on the operation that the user requested.

The remainder of this section will describe the FSMs for the four main operations in detail. The memory layout during each step of these operations will be shown as well.

4.7.1 CPA-Encryption

The CPA-secure encryption works as follows: First, the message to be encrypted is loaded in from the AMBA bus into the MSG_{CPA} area of the RAM. Then the FSM performs the encryption algorithm from Section 2.5.2. While doing so, it invokes four separate sub-FSMs for the sampling of polynomials, the NTT-operation, pointwise multiplication and addition of polynomials, and for the addition of the message to the error-polynomial $\mathbf{e}_3$. The result $(\mathbf{c}_1, \mathbf{c}_2)$ is stored at the memory addresses C1_{CPA} and C2_{CPA}, which can then be read by the user by requesting that memory area. In Figure 4.4, a visualization of the FSM is given. The numbers along the edges in that figure represent the order of operations.

Table 4.1 shows each step of the algorithm with the memory usage after each respective operation. The first column shows the operation, and the remaining columns show the respective memory locations, where the results are stored after each operation, using the notation from Section 4.4.

	$C1_{CPA}$	$C2_{CPA}$	$E1_{CPA}$	MSG_{CPA}
input msg	—	—	—	msg
$\mathbf{e}_1$ = random polynomial	—	—	$\mathbf{e}_1$	msg
$\tilde{\mathbf{e}}_1$ =NTT($\mathbf{e}_1$)	—	—	$\tilde{\mathbf{e}}_1$	msg
$\mathbf{e}_2$ = random polynomial	—	$\mathbf{e}_2$	$\tilde{\mathbf{e}}_1$	msg
$\tilde{\mathbf{e}}_2$ = NTT($\mathbf{e}_2$)	—	$\tilde{\mathbf{e}}_2$	$\tilde{\mathbf{e}}_1$	msg
$\tilde{\mathbf{c}}_1 = (\tilde{\mathbf{a}} * \tilde{\mathbf{e}}_1) + \tilde{\mathbf{e}}_2$	$\tilde{\mathbf{c}}_1$	$\tilde{\mathbf{e}}_2$	$\tilde{\mathbf{e}}_1$	msg
$\mathbf{e}_3$ = random polynomial	$\tilde{\mathbf{c}}_1$	$\mathbf{e}_3$	$\tilde{\mathbf{e}}_1$	msg
$\mathbf{e}_3 + msg$	$\tilde{\mathbf{c}}_1$	$\mathbf{e}_3 + msg$	$\tilde{\mathbf{e}}_1$	msg
NTT($\mathbf{e}_3 + msg$)	$\tilde{\mathbf{c}}_1$	NTT($\mathbf{e}_3 + msg$)	$\tilde{\mathbf{e}}_1$	—
$\mathbf{c}_2 = (\tilde{\mathbf{p}} * \tilde{\mathbf{e}}_1)$ + NTT($\mathbf{e}_3 + msg$)	$\tilde{\mathbf{c}}_1$	$\tilde{\mathbf{c}}_2$	—	—
output $\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2$	$\tilde{\mathbf{c}}_1$	$\tilde{\mathbf{c}}_2$	—	—

Table 4.1: RAM contents during CPA_{enc}

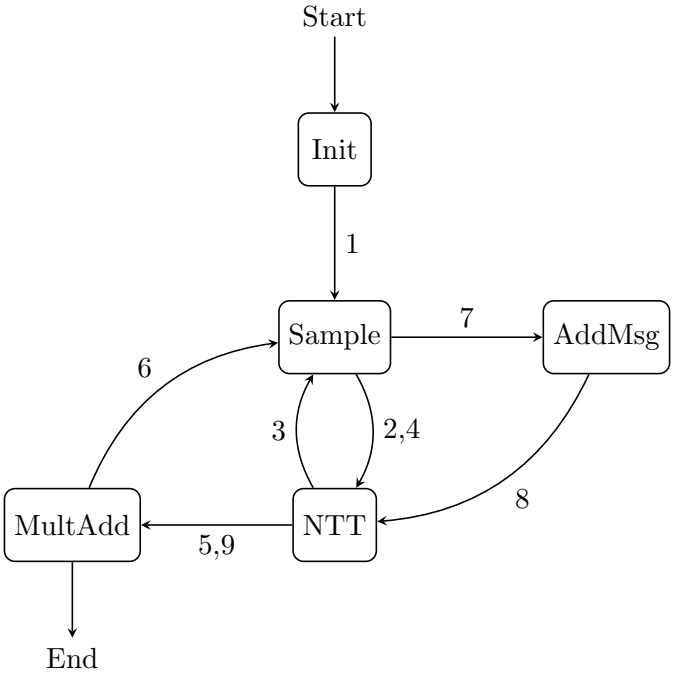


Figure 4.4: CPA_{enc} controller FSM

4.7.2 CPA-Decryption

After loading the input $(c1, c2)$ into the memory location C1_{CPA} and C2_{CPA}, the FSM performs the decryption by invoking FSMs for three sub-tasks: First, the multiplication and addition of polynomials $(\tilde{\mathbf{c}}_1 * \tilde{\mathbf{r}}_2 + \tilde{\mathbf{c}}_2)$, second, the INTT operation is called with the previous result as the input, and lastly, the result is decoded to recover the original message. These steps are also shown in the memory organization table for this operation in Table 4.2. Figure 4.5 shows a visualization of the FSM for the CPA_{dec} operation.

	$C1_{CPA}$	$C2_{CPA}$	$E1_{CPA}$	MSG_{CPA}
input $\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2$	$\tilde{\mathbf{c}}_1$	$\tilde{\mathbf{c}}_2$	—	—
$\tilde{\mathbf{c}}_1 * \tilde{\mathbf{r}}_2 + \tilde{\mathbf{c}}_2$	$\tilde{\mathbf{c}}_1 * \tilde{\mathbf{r}}_2 + \tilde{\mathbf{c}}_2$	—	—	—
$\mathbf{m\prime} = \text{INTT}(\tilde{\mathbf{c}}_1 * \tilde{\mathbf{r}}_2 + \tilde{\mathbf{c}}_2)$	$\mathbf{m\prime}$	—	—	—
$msg = \text{decode}(\mathbf{m\prime})$	$\mathbf{m\prime}$	—	—	msg
output msg	—	—	—	msg

Table 4.2: RAM contents during CPA_{dec}

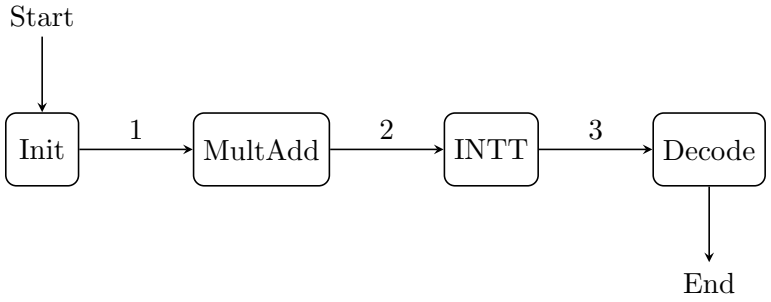


Figure 4.5: CPA_{dec} controller FSM

4.7.3 CCA-Encryption

For the CCA_{enc} operation, first the message to be encrypted, as well as a nonce v are loaded into the RAM locations MSG_{cpa} and MSG_{CCA}, respectively. Then, the nonce is hashed and XORed with the message, the result (c_3) is stored in the memory location C1_{CCA}. Afterwards, the PRNG is re-seeded with the hash of the nonce and the message $\text{H}(v||msg)$. The PRNG is re-seeded, such that the exact values for the error polynomials, generated during CPA_{enc}, can be reproduced later for the decryption. Once the PRNG is re-seeded, the nonce v is encrypted. The result of CPA_{enc} $(\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2)$ together with c_3 is the result. Table 4.3 shows these steps, together with the used memory locations. Figure 4.6 gives a visualization of the FSM for this operation.

	$C1_{CPA}$	$C2_{CPA}$	$E1_{CPA}$	$C1_{CCA}$	$C2_{CCA}$	MSG_{CPA}	MSG_{CCA}
input v, msg	—	—	—	—	—	v	msg
$\text{G}(v)$	SHAKE state: $\text{G}(v)$			—	—	v	msg
$c_3 = \text{G}(v) \oplus msg$	SHAKE state: $\text{G}(v)$			c_3	—	v	msg
$seed = \text{H}(v\|\|msg)$	SHAKE state: $\text{H}(v\|\|msg)$			c_3	—	v	msg
$\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2 = \text{CPA}_{\text{enc}}(v)$	$\tilde{\mathbf{c}}_1$	$\tilde{\mathbf{c}}_1$	—	c_3	—	—	—
output $\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2, c_3$	$\tilde{\mathbf{c}}_1$	$\tilde{\mathbf{c}}_1$	—	c_3	—	—	—

Table 4.3: RAM contents during CCA_{enc}

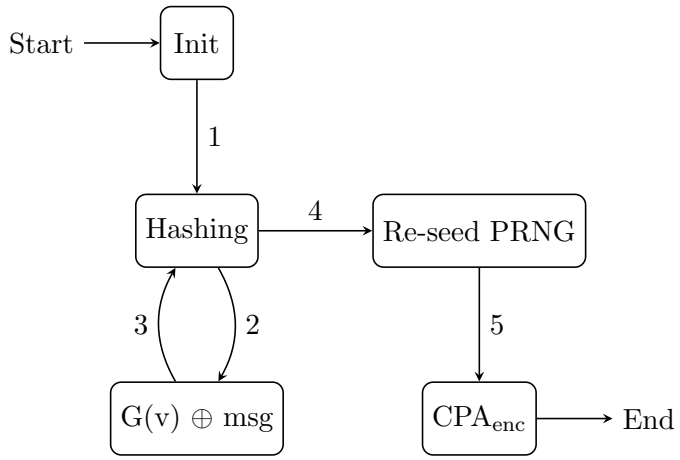


Figure 4.6: CCA_{enc} controller FSM

4.7.4 CCA-Decryption

The CCA_{dec} operation, works as follows. First the input $(\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2, c_3)$ is loaded into the memory locations C1_{CCA}, C2_{CCA}, MSG_{CCA}. Then, $(\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2)$ is decrypted and decoded to recover the nonce v. The nonce is then hashed and XORed with c_3, the result is the plaintext message. However, it is not immediately output. By hashing the nonce as well as the message, the seed is recovered, and the PRNG is re-seeded. Afterwards, the nonce is encrypted again. The result of that encryption $(\tilde{\mathbf{c}}_1\prime, \tilde{\mathbf{c}}_2\prime)$ is then compared with the input $(\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2)$. If these are equal, the message is output. If not, then the end user receivers an error flag, indicating that the operation could not be completed. This protects against adaptive chosen-ciphertext attacks.

Figure 4.7 shows a visualization of the FSM for the CCA_{dec} operation. Furthermore, Table 4.4 shows each step of the algorithm with the memory usage after each respective operation.

While the other operations (CPA_{dec}, CPA_{dec}, and CCA_{enc}) do not fully utilize the available RAM, during the hashing operations in the CCA2-secure decryption, all available memory locations are being used.

	$C1_{CPA}$	$C2_{CPA}$	$E1_{CPA}$	$C1_{CCA}$	$C2_{CCA}$	MSG_{CPA}	MSG_{CCA}
input $\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2, c_3$	—	—	—	$\tilde{\mathbf{c}}_1$	$\tilde{\mathbf{c}}_2$	—	c_3
$v = \text{CPA}_{\text{dec}}(\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2)$	—	—	—	$\tilde{\mathbf{c}}_1$	$\tilde{\mathbf{c}}_2$	v	c_3
$\text{G}(v)$	SHAKE state: $\text{G}(v)$			$\tilde{\mathbf{c}}_1$	$\tilde{\mathbf{c}}_2$	v	c_3
$msg = \text{G}(v) \oplus c_3$	SHAKE state: $\text{G}(v)$			$\tilde{\mathbf{c}}_1$	$\tilde{\mathbf{c}}_2$	v	msg
$seed = \text{H}(v\|\|msg)$	SHAKE state: $\text{H}(v\|\|msg)$			$\tilde{\mathbf{c}}_1$	$\tilde{\mathbf{c}}_2$	v	msg
$\tilde{\mathbf{c}}_1\prime, \tilde{\mathbf{c}}_2\prime = \text{CPA}_{\text{enc}}(v)$	$\tilde{\mathbf{c}}_1\prime$	$\tilde{\mathbf{c}}_2\prime$	—	$\tilde{\mathbf{c}}_1$	$\tilde{\mathbf{c}}_2$	—	msg
output msg if $\tilde{\mathbf{c}}_1\prime, \tilde{\mathbf{c}}_2\prime == \tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2$	$\tilde{\mathbf{c}}_1\prime$	$\tilde{\mathbf{c}}_2\prime$	—	$\tilde{\mathbf{c}}_1$	$\tilde{\mathbf{c}}_2$	—	msg

Table 4.4: RAM contents during CCA_{dec}

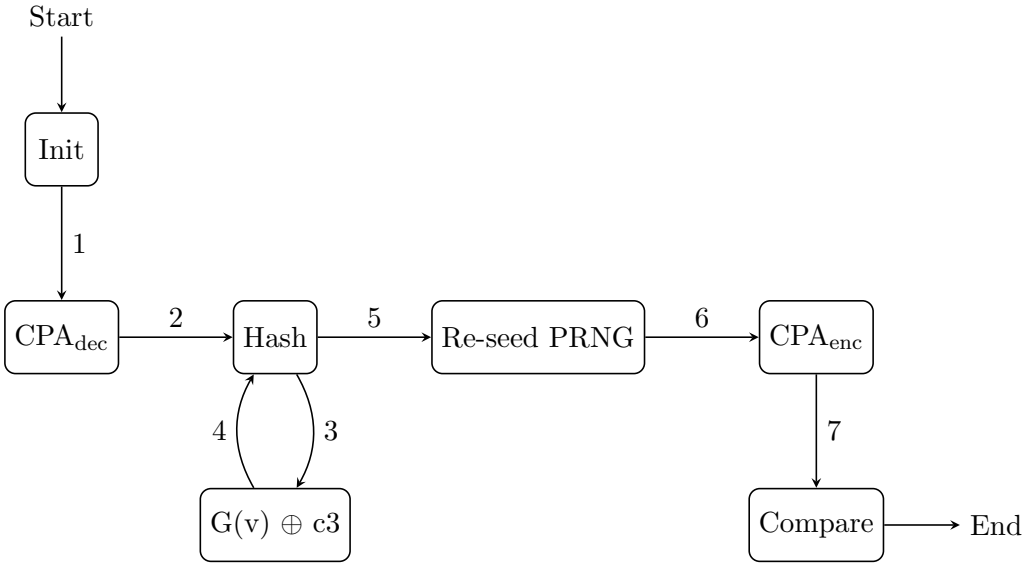


Figure 4.7: CCA$_{dec}$ controller FSM

4.7.5 Input and Output

The coprocessor is controlled by the user via the AMBA ABP bus. Apart from the four main operations for encryption and decryption shown above, the user can also re-seed the PRNG, as well as read from, and write to the memory locations as shown in Section 4.4.

Additionally, to serve as an evaluation platform for future implementation attacks, any other FSM can be invoked as well. For example, the implementation can be used to hash arbitrary data with the Keccak module, or to NTT-transform any polynomial that has previously been loaded into the RAM.

For practical applications, this behavior can be disabled, and strict memory checks can be enabled, to ensure that only some memory locations can be read and written, depending on the current operation.

Chapter 5

Results and Discussion

In this chapter, the implementation results are presented. First, the used tools and design flow is presented in Section 5.1. Detailed results for the area, power, and time requirements are then given in Section 5.2. Afterwards, Section 5.3 gives a comparison to related work. The results are then discussed, and an outlook for possible future work is given in Section 5.4. Finally, the results are summarized in Section 5.5.

5.1 Design Flow and Tools

Designing and implementing a hardware implementation is a complex process which involves many different steps. Various tools and description languages are required for this. After designing the ALU on paper, a high-level model was implemented using the JAVA programming language. The high-level model was refined until it was cycle accurate, and then used for the generation of test-vectors and configuration files. The test-vectors were later used for testing the hardware implementation. The high-level model additionally served as an evaluation platform for performance estimations and optimizations.

Based on the high-level model, the design was implemented using the hardware description language VHDL. For the memory, a single-port SRAM macro was provided by Faraday. The used standard-cell library is based on the low-leakage 65nm process technology by UMC. For synthesis, the Cadence Encounter™ RTL Compiler (version 14.2) was used. Place-and-Route, as well as the power simulation, was done by the Innovus® Implementation System (version 16.10).

5.2 Implementation Results

This section presents the detailed implementation results for area, power and time requirements.

5.2.1 Area Requirements

The area numbers, given by the RTL compiler, are expressed in μm^2 and gate equivalents (GE). One GE is equivalent to the size of a single 2-input NAND gate, which has an area of $1.44\mu m^2$ in the used UMC 65nm process. So multiplying the area in GE with 1.44 results in the area in μm^2.

The physical size is very dependent on the used manufacturing process and cannot be used for a fair comparison across different technology nodes. Using the number of GEs however, does allow for this comparison. It should be noted that GE is not an exact measurement. Synthesizing the design using a different tool flow or even different versions of the tools give different results.

The design was synthesized using many different parameter sets. To give a better understanding of the results, first, only the results from instantiations with the parameter set P_1 and support for CCA2-secure encryption will be discussed. Afterwards, the area requirements of all parameter sets are compared to each other. Finally, results for instantiations without support for the CCA2-secure algorithm are discussed.

Component	**Area**	
	[GE]	**[μm^2]**
ALU	3 186	4 588
SHA3	3 158	4 549
Sampler	590	851
Trivium	2 968	4 275
Other	2 817	4 051
Core Total	**12 719**	**18 314**
RAM	14 752	21 244
Total	**27 471**	**39 558**

Table 5.1: Area of components for parameter set P_1 (with CCA2 support).

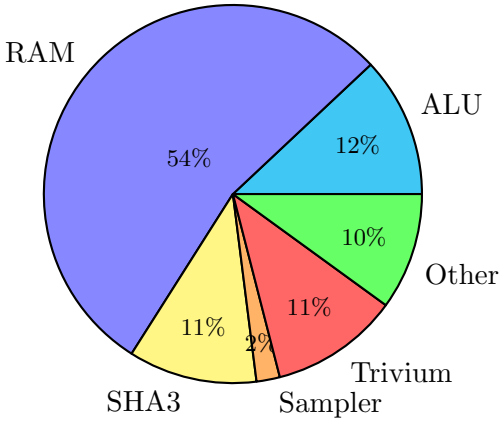


Figure 5.1: Area distribution for parameter set P_1 (with CCA2 support).

Table 5.1 shows a detailed listing of the area requirements for each component using the parameter set P_1 with support for CCA2-secure encryption and decryption. Figure 5.1 illustrates the relative area distribution using the same results. The total area for this instance is 27.4kGE or 39 558μm^2. Due to the large memory requirements of this scheme, over 50% of the total area is consumed by the RAM. The ALU is dominated by the big $\log_2(q)$-bit wide multiplier and amounts to 12% of the area, which is the same size as the hashing module. Trivium, used as PRNG, uses 11% of the area, and the sampler occupies only 2% of the total area. Summing up these components amounts to 90%, which leaves 10% for *Other*. This category includes the ROM, the AMBA interface, connections and multiplexers to all other components, the RAM-multiplexer and logic to switch the RAM access between the ALU and the hashing module, and most importantly the control-unit with all its FSMs, state registers, and counters. While it might be surprising that this category is so big, it can be explained by the many more registers compared to the ALU, and the many state machines.

Depending on the application, the RAM can be external to the core, so while it does consume more than half of the total area, it should not be considered as part of the core. In this work, the *core* is considered everything apart from the RAM.

If the core is instantiated without support for CCA2-secure encryption and decryption, it is much smaller as the hashing module would be gone entirely. The ALU, as well as the control unit, would also be much smaller and the RAM requirements would shrink by 25%. For the parameter set P_1, a total of 7.9kGE would be saved when synthesizing the design without support for CCA2-secure encryption/decryption.

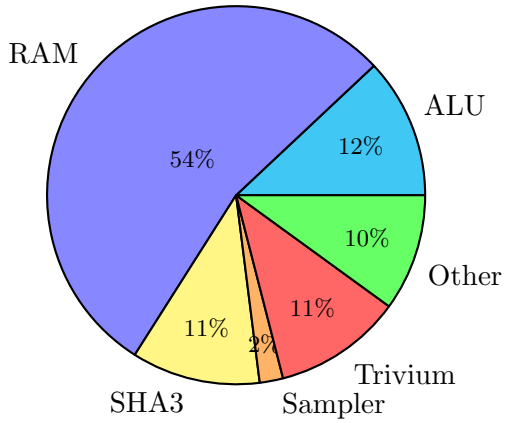


Figure 5.2: Area distribution for the parameter set P_1 ($n = 256$) with CCA2 support.

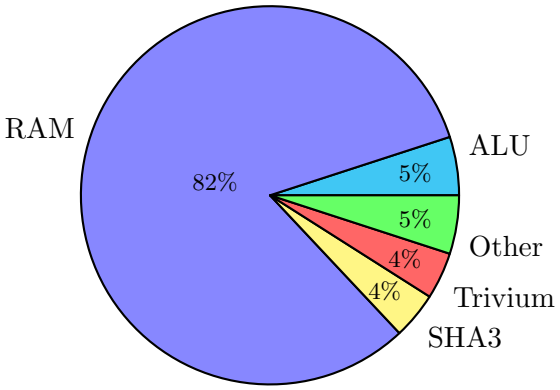


Figure 5.3: Area distribution for the parameter set P_5 ($n = 1024$) with CCA2 support.

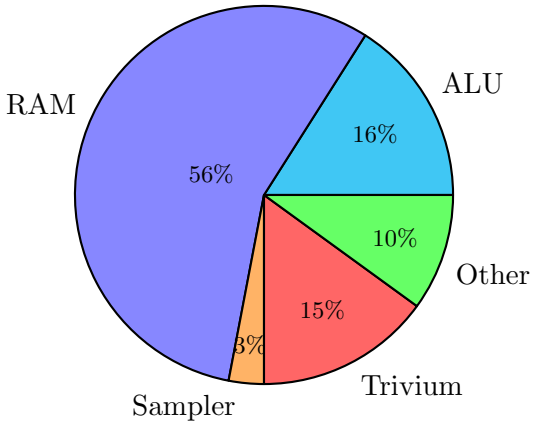


Figure 5.4: Area distribution for the parameter set P_1 ($n = 256$) without CCA2 support.

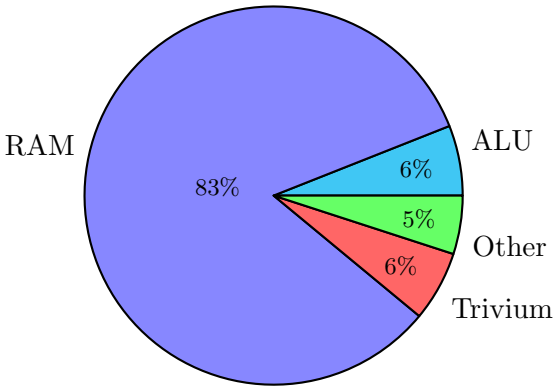


Figure 5.5: Area distribution for the parameter set P_5 ($n = 1024$) without CCA2 support.

Parameter Set	P_1	P_6	P_2	P_3	P_5	P_7	P_4
n	256	512	512	512	1 024	1 024	1 024
q	7 681	12 289	12 289	25 601	12 289	12 289	40 961
k	41	24	47	21	8	16	21
Area per component in GE							
ALU	3 186	3 514	3 514	3 759	3 514	3 514	4 147
SHA3	3 158	3 158	3 158	3 159	3 158	3 158	3 158
Sampler	590	416	615	596	155	406	630
Trivium	2 968	2 968	2 968	2 968	2 968	2 968	2 968
Other	2 817	2 991	2 994	2 989	3 344	3 074	3 031
Core Total	**12 719**	**13 047**	**13 249**	**13 471**	**13 139**	**13 120**	**13 934**
RAM	14 752	29 505	29 505	29 505	59 010	59 010	59 010
Total	**27 471**	**42 552**	**42 754**	**42 976**	**72 149**	**72 130**	**72 944**

Table 5.2: Area of components for all evaluated parameter sets (with CCA2 support). The first four rows describe the respective parameter set, while all other rows contain area values in GE.

Parameter Set	P_1	P_6	P_2	P_3	P_5	P_7	P_4
n	256	512	512	512	1 024	1 024	1 024
q	7 681	12 289	12 289	25 601	12 289	12 289	40 961
k	41	24	47	21	8	16	21
Area per component in GE							
ALU	3 076	3 395	3 395	3 569	3 397	3 397	3 790
Sampler	590	418	615	597	155	407	630
Trivium	2 968	2 968	2 968	2 968	2 968	2 968	2 968
Other	1 873	2 010	2 006	2 023	2 343	2 105	2 014
Core Total	**8 507**	**8 791**	**8 984**	**9 157**	**8 863**	**8 877**	**9 402**
RAM	11 007	22 015	22 015	22 015	44 030	44 030	44 030
Total	**19 514**	**30 806**	**30 999**	**31 172**	**52 893**	**52 907**	**53 432**

Table 5.3: Area of components for all evaluated parameter sets (without CCA2 support). The first four rows describe the respective parameter set, while all other rows contain area values in GE.

In Table 5.2, the area requirements for all evaluated parameter sets are enumerated. Figure 5.2 and Figure 5.3 show the relative area distributions for the two parameter sets P_1 (n = 256), and P_5 (n = 1024). Because the RAM requirements linearly rise with the parameter n, the set with n = 1024 has a RAM which is four times the size. This amounts to 82% of the total area, dwarfing any other component in size. Figure 5.4 and Figure 5.5 show the same area distributions, but for instantiations with support for the CCA2-secure algorithm. The relative area distributions are very similar to the previous figures. The only difference is the missing hashing module (SHA3) for instantiations without CCA2-support.

Comparing the core components for the smallest (P_1) and largest (P_4) instantiations, the core size only increases by about 9%. The size for the components Trivium and the hashing module stay the same because they are completely independent of the parameter set. Trivium would be bigger with more generated bits per cycle, however, in this work this variable is fixed to 16 bits per cycle across all parameter sets. The *Other* category also stays roughly the same, as it only has a few counters that are dependent on the variable n.

Compared to the other parameter sets, the size of the sampler for the set P_5 is much lower, while at the same time the value for *Other* increases by about the same amount that is missing from the sampler. In this case, the synthesizer could have moved parts of the sampler away from the component, as part of one of the optimization steps. It is also likely that the area increase for *Other* is unrelated because the area of the sampler should scale with the parameter k and the value for this parameter is the lowest across all sets.

Apart from the RAM, and the sampler for the set P_5, the only component that changes its size across the parameter sets is the ALU. This change is expected, as most parts (multiplier, registers) within that component scale with the size of $\log_2(q)$. However, this change is only very minor compared to the RAM scaling.

Table 5.3 shows the area of components with disabled support for CCA2-secure encryption. Compared to Table 5.2, the hashing module (SHA3) is completely gone, which means a reduction of 3.1kGE. Additionally, the ALU has decreased about 0.1 – 0.4 kGE because it no longer needs to handle the output from the hashing module or support an XOR operation. The *Other* part sees a reduction of 1kGE. This reduction is due to the missing glue logic to handle the hashing module and the FSMs, which were responsible for handling the actual CCA2-secure encryption and decryption. The most significant change, however, is the reduction in RAM area, which amounts to 25% (or 3.7 – 14.9kGE). In total, removing the support for CCA2-secure encryption can save 7.9 – 19.5 kGE (or 27 – 29%).

5.2.2 Timing Results

The timing numbers are given in the number of cycles a given operation takes to complete. At an operating frequency of 1 MHz, the cycle numbers are also equivalent to the runtime in μs.

The encryption and decryption times are constant and only depend on the chosen parameter set. Given a parameter set (n, k, q), the parameter n is the most significant with regards to the execution time, as it influences the time of almost all sub-operations. The parameter k only influences the runtime for the sampling of polynomials. The prime q does not have any timing impact at all. This is due to the reduction algorithm chosen in this work, which has a constant runtime for any choice of q.

Using the default parameter set P_1, a CPA-secure encryption takes 32 908 cycles, while the decryption takes only 12 168 cycles. When using the CCA2-secure variant, the numbers are much bigger. A CCA2-secure encryption, at 77 402 cycles, takes more than twice as long compared to the CPA-secure counterpart. The decryption also takes about twice the time at 64 205 cycles compared to 32 908 cycles for the CPA-secure variant.

The runtime of a CPA-secure encryption is the sum of the runtime of sampling three error polynomials ($3 \cdot 1\,792$ cycles), two multiplications and addition of polynomials ($2 \cdot 1\,538$ cycles), two forward NTT operations without bit-reversal ($2 \cdot 7\,771$ cycles), a single forward NTT operation with bit-reversal (8 388 cycles), and adding the message m to the error polynomial e_3 (528 cycles). The distribution of the runtime for a single encryption is depicted in Figure 5.6.

A CPA-secure decryption is much simpler. It only consists of one multiplication and addition of a polynomial (1 538 cycles), one inverse NTT operation (10 353 cycles), and a decoding operation (273 cycles). This runtime distribution is shown in Figure 5.7.

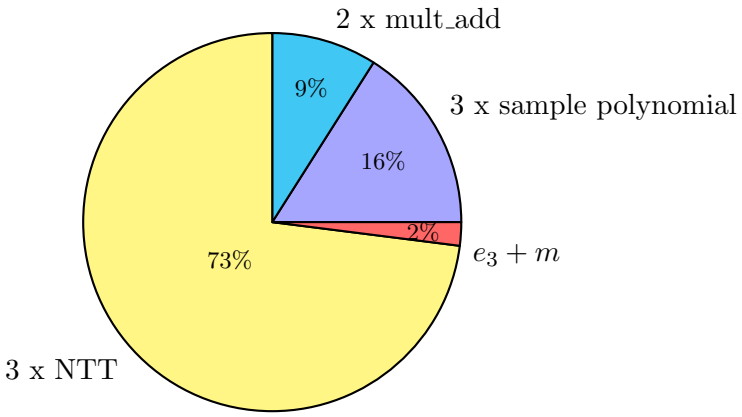


Figure 5.6: Distribution of the runtime for a CPA-secure encryption using the parameter set P_1

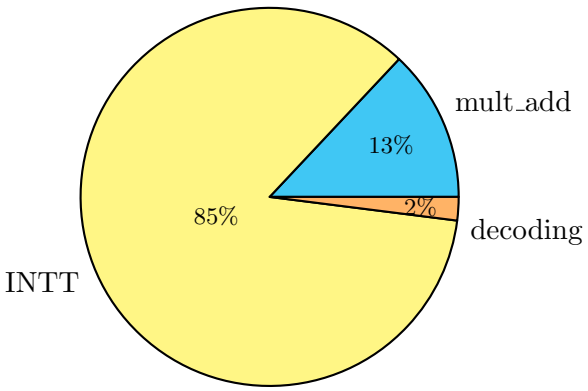


Figure 5.7: Distribution of the runtime for a CPA-secure decryption using the parameter set P_1

The runtime of the CCA2-secure encryption can be split up into two hashing operations (15 474 + 15 536 cycles), an XOR operation (145 cycles), re-seeding of the PRNG (58 cycles), a complete CPA-secure encryption (32 908 cycles), and a small state-transitioning overhead of 84 cycles. The runtime distribution for this operation is depicted in Figure 5.8. This figure clearly shows that the hashing operation (at 48%) is a significant part of the total runtime. 51% of the runtime is taken up by the actual CPA-secure encryption.

The decryption counterpart is comprised of a normal CPA-secure decryption (12 168 cycles) and encryption (32 908 cycles), two hashing operations (15 474 + 15 536 cycles), a comparison of two polynomials (514 cycles), and some smaller operations: decoding (273 cycles), XOR (145), re-seeding of the PRNG (58 cycles), and others (326 cycles). Figure 5.9 depicts the runtime distribution for this operation.

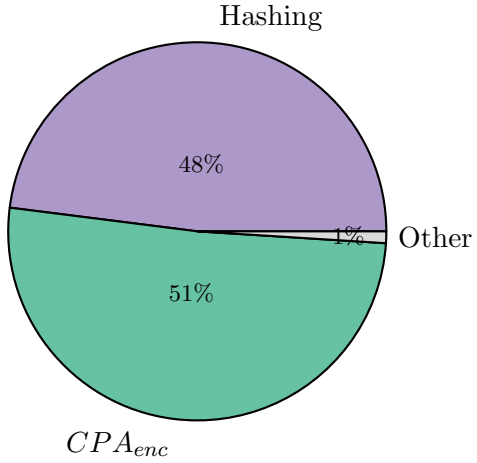


Figure 5.8: Distribution of the runtime for a CCA2-secure encryption using the parameter set P_1

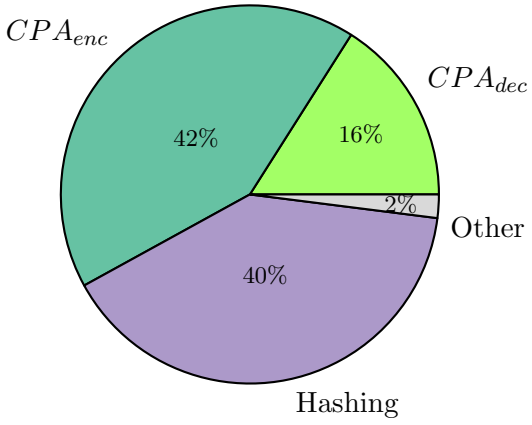


Figure 5.9: Distribution of the runtime for a CCA2-secure decryption using the parameter set P_1

For the parameter set P_1, each hashing operation takes 15.5k cycles. In comparison, the next biggest operation, the NTT operation, only takes about half the time at 8k cycles. The hashing operation takes up 40 – 48% of the total runtime. The specific Keccak implementation used in this work is optimized for low area. For added performance, a different implementation can be used that is optimized for performance, however this would significantly increase the area of this module. Another option could be to include a more lightweight hashing algorithm. However, this might negatively impact the security of the entire scheme and would need to be analyzed in detail.

Parameter Set	P_1	P_6	P_2	P_3	P_5	P_7	P_4
n	256	512	512	512	1 024	1 024	1 024
q	7 681	12 289	12 289	25 601	12 289	12 289	40 961
k	41	24	47	21	8	16	21
Cycles per operation							
hash XOR msg	145	289	289	289	577	577	577
decoding	273	545	545	545	1 089	1 089	1 089
G(v)	15 474	15 536	15 536	15 536	15 660	15 660	15 660
H(v || m)	15 536	15 660	15 660	15 660	31 409	31 409	31 409
mult_add	1 538	3 074	3 074	3 074	6 146	6 146	6 146
NTT forward	7 771	17 272	17 272	17 272	38 041	38 041	38 041
NTT forward with bit-reverse	8 388	18 505	18 505	18 505	40 554	40 554	40 554
NTT inverse with bit-reverse	10 353	22 490	22 490	22 490	48 603	48 603	48 603
bit-reverse	616	1 232	1 232	1 232	2 512	2 512	2 512
Scaling	2 049	4 097	4 097	4 097	8 193	8 193	8 193
PRNG re-seed from hash	58	58	58	58	58	58	58
PRNG re-seed external	14	14	14	14	14	14	14
comparison	514	1 026	1 026	1 026	2 050	2 050	2 050
$e_3 + m$	528	1 056	1 056	1 056	2 112	2 112	2 112
sample polynomial	1 792	2 048	3 584	2 048	2 048	3 072	4 096
$\boldsymbol{CPA_{dec}}$	**12 168**	**26 113**	**26 113**	**26 113**	**55 842**	**55 842**	**55 842**
$\boldsymbol{CPA_{enc}}$	**32 908**	**66 404**	**71 003**	**66 404**	**137 197**	**140 266**	**143 335**
$\boldsymbol{CCA_{dec}}$	**77 402**	**126 194**	**130 796**	**126 194**	**244 923**	**247 993**	**251 063**
$\boldsymbol{CCA_{enc}}$	**64 205**	**98 028**	**102 630**	**98 028**	**184 980**	**188 050**	**191 120**

Table 5.4: Number of cycles per operation for different parameter sets

Table 5.4 gives detailed runtime numbers for different operations and all evaluated parameter sets. The total runtime for the CPA-secure encryption and decryption almost perfectly scales with the dimension n, while the CCA2 counterpart does not. This is because the hashing operation stays the same for most parameter sets. Its runtime only depends on its internal block size and the length of v and m. For parameter sets with a dimension

of $n = 512$ or smaller (P_1, P_6, P_2 and P_3), the hashing operation has a much smaller runtime relative to other operations. The internal block size of the hashing module is 1344 bits, i.e. up to 1344 bits of data can be hashed in a single permutation. For the operation $H(v||m)$, $2n$ bits of data need to be hashed. Since $2n$ does not fit in a single block anymore for parameter sets with $n > 512$, two permutations are required instead. The sampling of error polynomials scales with the parameter k, while most other operations scale linearly with the dimension n. The only operation that has a constant runtime across all parameter sets is the seeding of the PRNG.

It should be noted, that the timing numbers for the inverse NTT operation do include the *Scaling* operation as well, even though it is listed separately.

Place-and-Route reported a maximum frequency of 70MHz for the instantiation of the parameter set P1 with support for CCA2-secure encryption. However, this number heavily depends on several factors like the used toolchain and compiler settings. Due to the high number of possible configurations, only one parameter set with a specific configuration was used to explore maximum frequency.

High clock speed was not the goal of this work, and embedded devices typically have a very low clock frequency. For example, Near-field communication (NFC) tags usually operate at a frequency of 13.56MHz or a fraction of that.

However, using a clock frequency of 70MHz would reduce the encryption time from 32 908μs to 470μs (or 64 205μs to 917μs for the CCA2-secure variant), and the decryption time from 12 168μs to 173μs (or 77 402μs to 1 105μs for the CCA2-secure variant).

5.2.3 Power Consumption

The average power consumption was determined for typical process and usage conditions. The power consumption values are given in μW/MHz and represent the total power consumption. Due to the use of a low-leakage process, the static power consumption is negligible, so the values are not split up into static and dynamic power.

With the default parameter set P_1, the implementation has an average power consumption during CCA2-secure encryption and decryption operations of 24.5μW at a clock frequency of 1MHz. Using CPA-secure operations, it is at 23.2μW/MHz.

Table 5.5 and Figure 5.10 show the average power consumption for different chip components during CCA2-secure encryption and decryption operations. Table 5.6 and Figure 5.11 show the same data, but for CPA-secure operations. Apart from the RAM and hashing module (SHA-3), the power consumption distribution roughly reflects the area requirements.

Table 5.7 and Table 5.8 show the power consumption of different chip components for the different parameter sets. The data from Table 5.8 is taken from an instantiation without support for CCA2-secure encryption and decryption. That is why that table shows a much lower power consumption for the RAM.

The total energy consumption can be computed by multiplying the power consumption with the runtime. This measurement is especially important for battery powered devices. CPA-secure encryption and decryption consume 764nJ and 282nJ of energy, whereas the CCA2-secure scheme uses 1 571nJ and 1 894nJ of energy, respectively.

Component	Power [μW/MHz]
ALU	6.29
SHA3	2.58
Sampler	0.44
Trivium	5.11
Other	4.63
Core Total	**19.05**
RAM	5.42
Total	**24.48**

Table 5.5: Average power consumption of components for CCA2-secure operations using the parameter set P_1

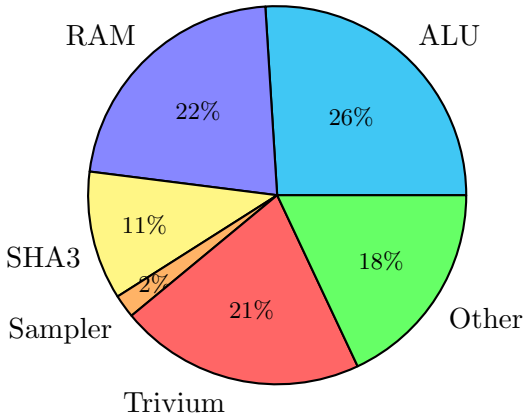


Figure 5.10: Power distribution for CCA2-secure operations using the parameter set P_1

Component	Power [μW/MHz]
ALU	9.44
Sampler	0.48
Trivium	4.90
Other	4.16
Core Total	**18.98**
RAM	4.24
Total	**23.23**

Table 5.6: Average power consumption of components for CPA-secure operations using the parameter set P_1

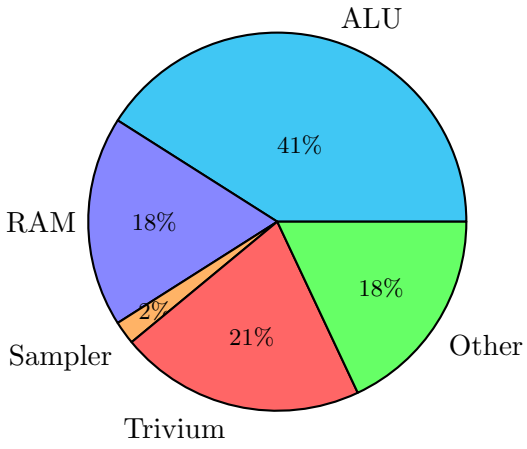


Figure 5.11: Power distribution for CPA-secure operations using the parameter set P_1

Parameter Set	P_1	P_6	P_2	P_3	P_5	P_7	P_4
n	256	512	512	512	1 024	1 024	1 024
q	7 681	12 289	12 289	25 601	12 289	12 289	40 961
k	41	24	47	21	8	16	21
Power consumption per component in μW/MHz							
ALU	6.3	7.6	7.2	9.7	8.2	8.2	12.0
SHA3	2.6	2.2	2.2	2.2	2.1	2.1	2.1
Sampler	0.4	0.4	0.5	0.4	0.0	0.3	0.5
Trivium	5.1	5.0	5.1	4.7	4.9	5.0	4.7
Other	4.6	5.0	4.8	5.5	5.4	5.3	5.6
Core Total	**19.1**	**20.2**	**19.9**	**22.5**	**20.7**	**20.9**	**24.8**
RAM	5.4	11.0	11.1	11.0	22.2	22.2	22.2
Total	**24.5**	**31.2**	**31.0**	**33.5**	**42.9**	**43.1**	**47.0**

Table 5.7: Average power consumption of components during CCA2-secure operations for different parameter sets. The first four rows describe the respective parameter set, while all other rows contain power consumption values in μW/MHz.

Parameter Set	P_1	P_6	P_2	P_3	P_5	P_7	P_4
n	256	512	512	512	1 024	1 024	1 024
q	7 681	12 289	12 289	25 601	12 289	12 289	40 961
k	41	24	47	21	8	16	21
Power consumption per component in μW/MHz							
ALU	9.4	10.0	9.0	12.3	9.8	9.6	14.2
Sampler	0.5	0.4	0.5	0.5	0.1	0.3	0.5
Trivium	4.9	4.7	5.3	4.7	5.0	5.0	4.8
Other	4.2	4.3	3.8	4.3	4.6	4.3	4.7
Core Total	**19.0**	**19.3**	**18.6**	**21.8**	**19.5**	**19.2**	**24.1**
RAM	4.2	8.5	8.5	8.5	17.0	17.0	17.0
Total	**23.2**	**27.8**	**27.1**	**30.3**	**36.5**	**36.2**	**41.1**

Table 5.8: Average power consumption of components during CPA-secure operations for different parameter sets. The first four rows describe the respective parameter set, while all other rows contain power consumption values in μW/MHz.

5.3 Comparison

This section compares the results of this thesis with other similar implementations. Because currently there exist no other ASIC implementations of Ring-LWE encryption schemes, first the results are compared to hardware implementations of other asymmetrical encryption schemes, then it is compared against FPGA implementations of various Ring-LWE schemes.

Implementation	**Process [nm]**	**Area [kGE]**	**Clock [MHz]**	**Scheme**	**Power [μW/MHz]**	**Time per operation [cycles (ms)]**
Kwon et al.[37]	500	156	50	RSA 1024		1 100 000 (22ms)
Liu et al.[43]	180	148	450	RSA 1024		2 105 352
Shieh et al.[73]	130	139	500	RSA 1024		2 105 344
Kuang et al.[36]	130	110	452	RSA 1024	89.16	
da Costa et al.[18]	180	107	125	RSA 1024		1 000 000 (8.44ms)
Huang and Wang[33]	130	5 100	333	RSA 12288		
Wenger and Hutter[82]	130	14.6		ECDSA $\mathbb{F}_{p192}$	39.54	1 394 000
Peßl and Hutter[61]	130	12.4		ECDSA $\mathbb{F}_{p160}$	42.42	139 930
This work (P_1)	65	19.5	70	Ring-LWE (CPA)	23.20	12 168 – 32 908 (0.17 – 1.1ms)
This work (P_1)	65	27.4	70	Ring-LWE (CCA2)	24.50	64 205 – 77 402 (0.17 – 1.1ms)

Table 5.9: Comparison of different hardware implementations of various asymmetrical encryption schemes.

Table 5.9 lists different low-resource ASIC implementations of asymmetrical encryption schemes. Some fields in this table were left empty because some details were not stated by the respective authors. It should be noted that for RSA 2–4kb keys are required nowadays, but such implementations were not available to compare with this work.

While comparing the area requirements of this work to RSA implementations shows a drastic reduction, comparing it to state of the art Elliptic Curve Digital Signature Algorithm (ECDSA) implementations paints a different picture. This work is about twice as big as optimized ECDSA implementations. However, when considering the power consumption and the runtime, it shows that this implementation is quite competitive. Both the power consumption and the runtime in cycles are lower than any of the other implementations.

Comparing Ring-LWE against other encryption schemes is a difficult task. For a fair comparison, the manufacturing process and security level should be the same, and complete numbers for power consumption, area, operation time, and clock frequency should be given. Due to the incomplete data present, it is not possible to objectively compare these different implementations in a fair manner. However, the presented data show that Ring-LWE-based encryption schemes can be as practical as other encryption schemes.

Implementation	Parameter Set (n, p, $\sigma \cdot \sqrt{2\pi}$)	Device	LUTs/FFs/DSPs/BRAM18	Clock [MHz]	Cycles (enc/dec)
Roy et al.[71]	P_1: (256, 7681,11.32)	V6LX75T	1 349 / 860 / 1 / 2	313	6.3k / 2.8k
Roy et al.[71]	P_2: (512,12289,12.18)	V6LX75T	1 536 / 953 / 1 / 3	278	13.3k / 5.8k
Pöppelmann et al.[64]	P_1: (256, 7681,11.32)	V6LX75T	4 549 / 3 624 / 1 / 12	262	6.8k / 4.4k
Pöppelmann et al.[64]	P_2: (512,12289,12.18)	V6LX75T	5 595 / 4 760 / 1 / 14	251	13.7k / 8.8k
Pöppelmann et al.[65]	(256, 4096, 8.35)	S6LX9	317 / 238 / 95 / 1	144	136.2k / –
Pöppelmann et al.[65]	(256, 4096, 8.35)	S6LX9	112 / 87 / 32 / 1	189	– / 66.3k
Göttert et al.[29] (encrypt)	P_1: (256, 7681,11.32)	V6LX240T	298 016 / 143 396 / ? / ?		
Göttert et al.[29] (decrypt)	P_1: (256, 7681,11.32)	V6LX240T	124 158 / 65 174 / ? / ?		
This work	P_1: (256, 7681,11.32)	–	– / – / – / –	70	32.9k / 12.1k
This work	P_2: (512,12289,12.18)	–	– / – / – / –		71.0k / 26.1k

Table 5.10: Comparison of different FPGA implementations of Ring-LWE-based encryption schemes.

Table 5.10 lists different FPGA implementations of Ring-LWE-based encryption schemes. Many FPGA implementations use either the parameter set P_1 or P_2, which is very convenient for comparing them to each other. All cycle numbers in this table use CPA-secure schemes.

When comparing the number of cycles an encryption or decryption takes, it is immediately noticeable that the presented implementation needs significantly longer. The use of different RAMs can largely explain the longer runtime. Most FPGA implementations use a high number of (dual-port) block RAM modules, whereas this implementation relies on a single instance of a single-port SRAM. A dual-port RAM typically allows reading or writing from/to two different addresses at the same time, while a single-port RAM only allows one such access in each cycle. Having multiple RAM modules additionally increase the number of simultaneous accesses. The limiting factor in most Ring-LWE implementations is how fast memory can be read and written. Being able to write or read multiple words simultaneously allows for much faster implementations. That being said, this implementation is still competitive even though it is not optimized for performance.

5.4 Discussion and Future Work

In this section, several points for discussion and possible future work are presented.

Tighter integration of external blocks.
In this work, an external implementation of the Keccak (SHA-3) algorithm was integrated for its hashing functionality. While the internals of this block were not studied in detail, it is estimated that 100 to 500 GE could be saved with tighter integration and sharing of resources like internal registers. However, this comes at the cost of lost flexibility. The current design allows exploring different PRNGs and hashing algorithms due to its relatively loose coupling. This advantage is lost by integrating these blocks more tightly.

Area optimization.
This work aimed to provide a 'low-resource' implementation, which includes low area, low power, and low runtime. However, certain trade-offs have been made in regards to the area, to allow for a simpler implementation or a faster runtime. When aiming for a low area, there are several places that can be optimized.

Firstly, the number of bits that the PRNG generates can be lowered. This makes the PRNG block up to 1kGE smaller, at the cost of a higher runtime for the sampling of polynomials. Another possibility would be to remove the PRNG block entirely and rely on the hashing module to generate pseudo-random numbers. This was initially planned for this work, but it turned out to be impractical due to the increased runtime requirements. However, if the runtime does not matter and the area has to be kept at a minimum, it may still be a viable choice to use the hashing module as a PRNG. Furthermore, trade-offs can be explored by using different Keccak implementations. Additionally, instead of Keccak, different hashing algorithms can be used. However using a less secure hashing algorithm could potentially weaken the encryption scheme.

Secondly, for some applications, it could make sense to store the keys in a ROM instead of the RAM, which would significantly cut down on the area for that RAM. Since the keys make up 49% of the RAM words, roughly an equal relative amount of area would be saved from the RAM.

Thirdly, the multiplier in the ALU is one of the biggest parts in this implementation because it is a full-width multiplier which only takes a single cycle for most multiplications. Because the area requirements of a multiplier usually increase with the square of the input width, reducing it to half the width would save a considerable amount of area. However, additional pipelining, as well as a different reduction algorithm are most likely needed to keep the performance at an acceptable level.

Lastly, the *Other* part, discussed in Section 5.2.1, make up a substantial portion of the total area requirements. However, the used tools do not provide insight as to the exact reason for this. Additionally, some modules, like the ROM, were merged with other blocks during the compilation. In any case, this needs to be investigated, as there is most likely much potential for improvement.

Performance optimization.
High performance was not the goal of this implementation, which leaves many areas for improvement in this regard. Additional pipelining and an optimized reduction algorithm, could redcude the total runtime of an encryption or decryption.

During a normal CPA-secure encryption, the sampling of the error polynomial e_3 could be done just-in-time and be combined with the step of adding the message m to e_3. Additionally, for the steps, where an error polynomial is directly NTT-transformed, there is no need to store the polynomial in the RAM and then perform the NTT separately. This could also be combined, such that the noise is sampled just in time during the NTT operation. While this would increase the complexity of the state machines slightly, it would also decrease the number of cycles for encryption.

Added functionality.
During the implementation of this work, a corrected version of the CCA2-secure scheme was presented [53], however, it has not been implemented in this work. These corrections should be implemented as part of any future work.

Additionally, different encodings, such that the message size does not have to be equal to the dimension n could be implemented. For example, Oder et al. use a 4:1 encoding, where each message bit gets encoded to four bits [52]. While this decreases the number of bits that can be encrypted in one run, it also decreases the chance of decoding errors significantly.

Additional schemes, like the Key Encapsulation Mechanism (KEM) scheme 'Kyber' presented in [11] could be added with minimal effort because it shares many similarities with the already implemented schemes.

Integration with other devices.
This implementation was designed as a coprocessor, so that it can be integrated with many other works. When integrating this work in RFID/NFC tags, unless the target implementation already uses an AMBA bus, additional logic for the wireless communication would be needed. For microprocessors which already have an (S)RAM, only the core would be needed, and the RAM could be shared between the microprocessor and this implementation. This would reduce the area requirements significantly, as the RAM is one of the largest parts in this work.

SCA-strengthening.
As previously mentioned, this work has a constant runtime, which is one important aspect of strengthening against side-channel attacks (SCA). However, many more aspects need to be considered for full protection against side-channel attacks. For instance, it is likely to be easy to attack this work using differential power analysis (DPA), because no masking or hiding was implemented. Other works, like the scheme presented by Oder et al., already consider such attack vectors [52]. Recent work by Primas et al. uses a power analysis attack to recover the full secret key [67]. It is suspected that this kind of attack would also work on this implementation, so additional countermeasures must be implemented before this work can be used in practice.

However, properly securing an implementation is no easy task. Every part of the implementation needs to be secured. For this implementation, this means that the the ALU, and the sampler, as well the integrated blocks (Keccak and Trivium) need to be analyzed and secured as well.

5.5 Summary

In this chapter, detailed implementation results have been presented. Using a manufacturing process with 65nm, the area requirements are between 27.4 and 72.9kGE (39 558– 105 039μm^2), depending on the used parameter set. When instantiating without support for CCA2 security, the area requirements are only 19.5– 53.4kGE (28 100– 76 942μm^2).

A CPA-secure encryption takes between 32 908 and 143 335 cycles. In comparison, a CPA-secure decryption only takes 12 168 – 55 842 cycles. Using the CCA2-secure scheme, encryptions take 64 205 – 191 120 cycles, and decryptions 77 402 – 251 063 cycles. At an operating frequency of 70MHz and using the parameter set P_1, encryptions can be as quick as 470μs and decryptions only take 173μs. For the CCA2-secure variant, these numbers increase to 917μs and 1 105μs, respectively.

The power consumption is only between 23 and 47μW/MHz, which is low enough to be suitable for passively-powered devices like NFC tags.

The design was compared to similar implementations, and it performs very well, considering its focus on low-resource requirements. However, many areas of this work still offer room for improvement.

Chapter 6

Conclusions

In this thesis, the design of a low-resource ASIC implementation of a lattice-based encryption scheme was presented. After defining the encryption schemes, the requirements and basic design choices for this implementation was discussed. Then the design was presented, and the results were discussed in detail.

The design uses several state-of-the-art optimizations, like integrating the scaling within the NTT operation and precomputing the so-called twiddle factors. For the modular reduction, the Barrett reduction algorithm was chosen because of its minimal impact on the area requirements. Thanks to the ALU design, the modular reduction algorithm requires no additional logic apart from an FSM. In addition to the CPA-secure scheme, a CCA2-secure encryption scheme was implemented. This requires the use of a hashing function, for this purpose an existing implementation of the Keccak (SHA-3) algorithm was integrated. For the generation of error polynomials, a binomial sampler was implemented instead of using a Gaussian sampler. This significantly reduces the complexity and area requirements without impacting the security of the scheme. Trivium, a very small and configurable stream cipher, was integrated to act as a PRNG for the sampler. For the main memory, a single-port SRAM macro was used, which is much smaller than a dual-port macro.

To counter side-channel attacks, the entire implementation runs in constant and data-independent time. Because there exists no standardized parameter set, the implementation was designed to have flexible parameters.

The results were evaluated using many different parameter sets, with lattice dimensions between 256 and 1024. These parameter sets have been previously proposed. Often claming between 128 and 256 bits. Instantiations for both the CPA-secure as well as the CCA2-secure variant were tested.

The implementation results show that a CCA2-secure Ring-LWE-based encryption scheme can be implemented very efficiently. The implementation was synthesized using the low leakage 65nm manufacturing process from UMC. After synthesis, the area requirements are 27.4kGE (39 558μm^2), from which 14.7kGE (21 242μm^2) is used by the RAM. When instantiating without support for CCA2 security, the area requirements are as low as 19.5kGE or 28 100μm^2. This area reduction of the core is mainly due to the 3kGE-sized hashing module, which only exists in the CCA2-secure variant.

Depending on the used parameter set, a CPA-secure encryption takes between 32 908 and 143 335 cycles. In comparison, a CPA-secure decryption only takes between 12 168 and 55 842 cycles.

Using the CCA2-secure scheme, encryptions take between 64 205 and 191 120 cycles and decryptions between 77 402 and 251 063. At an operating frequency of 70MHz, encryptions can be as quick as 470μs and decryptions only take 173μs. For the CCA2-secure variant, it is 917μs and 1 105μs respectively. Apart from the hashing in the CCA2-secure scheme, the NTT takes up the largest portion of the total runtime.

The power consumption is only between 23 and 47μW/MHz, which is low enough to be suitable for passively-powered devices like NFC tags.

This first ASIC implementation of a lattice-based encryption scheme was compared to similar implementations. The comparison shows that this implementation is practical, and competes directly with ASIC designs of other asymmetrical encryption schemes. However, it was also shown that there still exist areas for improvement depending on the desired target application. This implementation can serve as a base to build upon for future implementations. Thanks to its simple design, it can be used as an evaluation platform for lattice-based attacks and to implement future ideal lattice-based schemes.

Abbreviations

ALU Arithmetic logic unit. 24, 26, 28–33, 37, 38, 45–47, 49, 58, 59, 61

AMBA Advanced Microcontroller Bus Architecture. 23, 28, 30, 31, 39, 40, 44, 46, 59

APB Advanced Peripheral Bus. 28

ASIC Application-Specific Integrated Circuit. iv, v, 2, 21–23, 56, 61, 62

BLISS Bimodal Lattice Signature Scheme. 20

CCA2 Adaptive Chosen-Ciphertext Attack. iv, 3, 16, 17, 23, 25, 29, 36–39, 43, 46–56, 59, 61, 62

CDT Cumulative Distribution Table. 15

CMOS Complementary Metal-Oxide-Semiconductor. 22

CPA Chosen-Plaintext Attack. iv, 17, 36, 39, 50–59, 61

CVP Closest Vector Problem. 6

DPA differential power analysis. 59

ECC Elliptic Curve Cryptography. 1, 2, 5, 7, 13, 16, 22, 28

ECDSA Elliptic Curve Digital Signature Algorithm. 56

FFT Fast Fourier Transform. 3, 10, 11

FPGA Field Programmable Gate Array. iv, v, 1, 19, 20, 22, 23, 56, 57

FSM Finite-state Machine. 30, 34, 38–44, 46, 49, 61

GE gate equivalents. iv, v, 26, 45–49, 57–59, 61

IFFT Inverse Fast Fourier Transform. 10

INTT Inverse Number Theoretic Transform. 10, 11, 13, 34, 41

KEM Key Encapsulation Mechanism. 59

LWE Learning With Errors Problem. 5–9, 13, 20

NFC Near-field communication. iv, v, 53, 59, 60, 62

NIST National Institute of Standards and Technology. 25

NSA National Security Agency. 1

NTT Number Theoretic Transform. iv, v, 3, 8, 10, 11, 13, 19, 20, 23, 24, 27, 28, 30, 32, 34, 40, 44, 50, 52, 53, 58, 61, 62

PAR Place-and-Route. 53

PRNG Pseudorandom Number Generator. iv, 17, 20, 23, 25, 26, 28, 30, 37–39, 42–44, 46, 51, 53, 57, 58, 61

RAM Random Access Memory. iv, v, 20, 23, 24, 26, 28–34, 36–44, 46, 47, 49, 53, 57–59, 61

RFID Radio-frequency identification. 22, 59

Ring-LWE Ring-Learning With Errors. iv, v, 1, 7–9, 13–17, 20, 22–24, 26, 27, 29, 32, 56, 57, 61

ROM Read-only Memory. 28, 30, 34, 36, 46, 58

RSA Rivest-Shamir-Adleman Cryptosystem. 1, 2, 5, 7, 16, 22, 28, 56

SCA Side-Channel Attack. 59

SHAKE SHAKE - A variant of the Keccak Hashing Algorithm. 26

SoC System on Chip. 22

SRAM static random access memory. iv, v, 23, 45, 61

SVP Shortest Vector Problem. 5–7

Bibliography

[1] S. Agrawal, D. Boneh, and X. Boyen. Efficient lattice (H)IBE in the standard model. In *Proc. of Eurocrypt'10*, volume 6110 of *LNCS*, pages 553–572, 2010. URL https://doi.org/10.1007/978-3-642-13190-5_28.

[2] M. Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 99–108, New York, NY, USA, 1996. ACM. ISBN 0-89791-785-5. doi: 10.1145/237814.237838. URL http://doi.acm.org/10.1145/237814.237838.

[3] A. Akavia, S. Goldwasser, and V. Vaikuntanathan. *Simultaneous Hardcore Bits and Cryptography against Memory Attacks*, pages 474–495. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-00457-5. doi: 10.1007/978-3-642-00457-5_28. URL http://dx.doi.org/10.1007/978-3-642-00457-5_28.

[4] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange—a new hope. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 327–343, Austin, TX, 2016. USENIX Association. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim.

[5] B. Applebaum, D. Cash, C. Peikert, and A. Sahai. *Fast Cryptographic Primitives and Circular-Secure Encryption Based on Hard Learning Problems*, pages 595–618. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-03356-8. doi: 10.1007/978-3-642-03356-8_35. URL http://dx.doi.org/10.1007/978-3-642-03356-8_35.

[6] ARM. Amba apb protocol version: 2.0 - specification, 2010. URL http://infocenter.arm.com/help/topic/com.arm.doc.ihi0024-/.

[7] A. Aysu, C. Patterson, and P. Schaumont. Low-cost and area-efficient fpga implementations of lattice-based cryptography. In *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*, pages 81–86. IEEE, 2013.

[8] S. Bai and S. D. Galbraith. An improved compression technique for signatures based on learning with errors. In *Cryptographers Track at the RSA Conference*, pages 28–47. Springer, 2014.

[9] S. Bai, A. Langlois, T. Lepoint, D. Stehlé, and R. Steinfeld. Improved security proofs in lattice-based cryptography: using the rényi divergence rather than the statistical distance. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 3–24. Springer, 2015.

[10] J. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from lwe. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1006–1018. ACM, 2016.

[11] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, and D. Stehlé. Crystals–kyber: a cca-secure module-lattice-based kem. *IACR Cryptology ePrint Archive*, 2017:634, 2017.

[12] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. Post-quantum key exchange for the tls protocol from the ring learning with errors problem. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 553–570. IEEE, 2015.

[13] M. Braithwaite. Experimenting with post-quantum cryptography. Google Security Blog, 2016. `https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html`.

[14] J. Buchmann, F. Göpfert, T. Güneysu, T. Oder, and T. Pöppelmann. High-performance and lightweight lattice-based public-key encryption. In *Proceedings of the 2nd ACM International Workshop on IoT Privacy, Trust, and Security*, pages 2–9. ACM, 2016.

[15] D. Cash, D. Hofheinz, E. Kiltz, and C. Peikert. *Bonsai Trees, or How to Delegate a Lattice Basis*, pages 523–552. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-13190-5. doi: 10.1007/978-3-642-13190-5_27. URL `http://dx.doi.org/10.1007/978-3-642-13190-5_27`.

[16] D. D. Chen, N. Mentens, F. Vercauteren, S. S. Roy, R. C. Cheung, D. Pao, and I. Verbauwhede. High-speed polynomial multiplication architecture for ring-lwe and she cryptosystems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 62 (1):157–166, 2015.

[17] H.-C. Chen and Y. Asau. On generating random variates from an empirical distribution. *AIIE Transactions*, 6(2):163–166, 1974.

[18] C. A. da Costa, R. L. Moreno, O. S. Carpinteiro, and T. C. Pimenta. A 1024 bit rsa coprocessor in cmos. In *Microelectronics (ICM), 2013 25th International Conference on*, pages 1–4. IEEE, 2013.

[19] N. S. Dattani and N. Bryans. Quantum factorization of 56153 with only 4 qubits. *CoRR*, abs/1411.6758, 2014.

[20] R. de Clercq, S. S. Roy, F. Vercauteren, and I. Verbauwhede. Efficient software implementation of ring-lwe encryption. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 339–344, San Jose, CA, USA, 2015. EDA Consortium. ISBN 978-3-9815370-4-8. URL `http://dl.acm.org/citation.cfm?id=2755753.2755830`.

[21] Y. Dodis, S. Goldwasser, Y. Tauman Kalai, C. Peikert, and V. Vaikuntanathan. *Public-Key Encryption Schemes with Auxiliary Inputs*, pages 361–381. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-11799-2. doi: 10.1007/978-3-642-11799-2_22. URL `http://dx.doi.org/10.1007/978-3-642-11799-2_22`.

[22] L. Ducas and P. Q. Nguyen. *Faster Gaussian Lattice Sampling Using Lazy Floating-Point Arithmetic*, pages 415–432. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-34961-4. doi: 10.1007/978-3-642-34961-4_26. URL `http://dx.doi.org/10.1007/978-3-642-34961-4_26`.

[23] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. Lattice signatures and bimodal gaussians. In *Advances in Cryptology–CRYPTO 2013*, pages 40–56. Springer, 2013.

[24] E. Fujisaki and T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual International Cryptology Conference*, pages 537–554. Springer, 1999.

[25] S. D. Galbraith, C. Petit, B. Shani, and Y. B. Ti. On the security of supersingular isogeny cryptosystems. In *ASIACRYPT (1)*, pages 63–91. Springer, 2016. doi: 10.1007/978-3-662-53887-6_3. URL `https://eprint.iacr.org/2016/859`.

[26] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-506-2. doi: 10.1145/1536414.1536440. URL `http://doi.acm.org/10.1145/1536414.1536440`.

[27] C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, pages 197–206, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-047-0. doi: 10.1145/1374376.1374407. URL `http://doi.acm.org/10.1145/1374376.1374407`.

[28] Google. A preview of bristlecone, googles new quantum processor, 2018. URL `https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html`.

[29] N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. Huss. *On the Design of Hardware Building Blocks for Modern Lattice-Based Encryption Schemes*, pages 512–529. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-33027-8. doi: 10.1007/978-3-642-33027-8_30. URL `http://dx.doi.org/10.1007/978-3-642-33027-8_30`.

[30] L. Groot Bruinderink, A. Hülsing, T. Lange, and Y. Yarom. Flush, gauss, and reload – a cache attack on the bliss lattice-based signature scheme. In B. Gierlichs and A. Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, pages 323–345, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-53140-2.

[31] T. Güneysu, V. Lyubashevsky, and T. Pöppelmann. *Practical Lattice-Based Cryptography: A Signature Scheme for Embedded Systems*, pages 530–547. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-33027-8. doi: 10.1007/978-3-642-33027-8_31. URL `http://dx.doi.org/10.1007/978-3-642-33027-8_31`.

[32] T. Györfi, O. Cret, G. Hanrot, and N. Brisebarre. High-throughput hardware architecture for the swifft/swifftx hash functions. *IACR Cryptology ePrint Archive*, 2012: 343, 2012.

[33] X. Huang and W. Wang. A novel and efficient design for an rsa cryptosystem with a very large key size. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62 (10):972–976, 2015.

[34] H. B. Keller and J. R. Swenson. Experiments on the lattice problem of gauss. *Mathematics of Computation*, 17(83):223–230, 1963.

[35] D. Knuth. The complexity of nonuniform random number generation. *Algorithm and Complexity, New Directions and Results*, pages 357–428, 1976.

[36] S.-R. Kuang, J.-P. Wang, K.-C. Chang, and H.-W. Hsu. Energy-efficient high-throughput montgomery modular multipliers for rsa cryptosystems. *IEEE Transactions on very large scale integration (VLSI) systems*, 21(11):1999–2009, 2013.

[37] T.-W. Kwon, C.-S. You, W.-S. Heo, Y.-K. Kang, and J.-R. Choi. Two implementation methods of a 1024-bit rsa cryptoprocessor based on modified montgomery algorithm. In *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, volume 4, pages 650–653. IEEE, 2001.

[38] J. L. Lagrange. Recherches darithmetique. nouveaux memoires de lacademie de berlin. 1773.

[39] P. L'Ecuyer. *Non-uniform Random Variate Generations*, pages 991–995. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-04898-2. doi: 10.1007/978-3-642-04898-2_408. URL `https://doi.org/10.1007/978-3-642-04898-2_408`.

[40] T. Lepoint and M. Naehrig. *A Comparison of the Homomorphic Encryption Schemes FV and YASHE*, pages 318–335. Springer International Publishing, Cham, 2014. ISBN 978-3-319-06734-6. doi: 10.1007/978-3-319-06734-6_20. URL `http://dx.doi.org/10.1007/978-3-319-06734-6_20`.

[41] R. Lindner and C. Peikert. Better key sizes (and attacks) for lwe-based encryption. In *Proceedings of the 11th International Conference on Topics in Cryptology: CT-RSA 2011*, CT-RSA'11, pages 319–339, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19073-5. URL `http://dl.acm.org/citation.cfm?id=1964621.1964651`.

[42] M. Liu and P. Q. Nguyen. *Solving BDD by Enumeration: An Update*, pages 293–309. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-36095-4. doi: 10.1007/978-3-642-36095-4_19. URL `http://dx.doi.org/10.1007/978-3-642-36095-4_19`.

[43] Q. Liu, F. Ma, D. Tong, and X. Cheng. A regular parallel rsa processor. In *Circuits and Systems, 2004. MWSCAS'04. The 2004 47th Midwest Symposium on*, volume 3, pages iii–467. IEEE, 2004.

[44] Z. Liu, H. Seo, S. S. Roy, J. Großschädl, H. Kim, and I. Verbauwhede. Efficient ring-lwe encryption on 8-bit avr processors. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 663–682. Springer, 2015.

[45] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. *SWIFFT: A Modest Proposal for FFT Hashing*, pages 54–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-71039-4. doi: 10.1007/978-3-540-71039-4_4. URL `http://dx.doi.org/10.1007/978-3-540-71039-4_4`.

[46] V. Lyubashevsky, C. Peikert, and O. Regev. *On Ideal Lattices and Learning with Errors over Rings*, pages 1–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-13190-5. doi: 10.1007/978-3-642-13190-5_1. URL `http://dx.doi.org/10.1007/978-3-642-13190-5_1`.

[47] D. Micciancio and S. Goldwasser. *Complexity of Lattice Problems: a cryptographic perspective*, volume 671 of *The Kluwer International in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, Massachusetts, Mar. 2002.

[48] D. Micciancio and O. Regev. Lattice-based cryptography. In *Post-quantum cryptography*, pages 147–191. Springer, 2009.

[49] NSA/IAD. CNSA Suite and Quantum Computing FAQ, January 2016. `https://www.iad.gov/iad/library/ia-guidance/ia-solutions-for-classified/algorithm-guidance/cnsa-suite-and-quantum-computing-faq.cfm`.

[50] T. Oder, T. Pöppelmann, and T. Güneysu. Beyond ecdsa and rsa: Lattice-based digital signatures on constrained devices. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6. ACM, 2014.

[51] T. Oder, T. Güneysu, F. Valencia, A. Khalid, M. O'Neill, and F. Regazzoni. Lattice-based cryptography: From reconfigurable hardware to asic. In *Integrated Circuits (ISIC), 2016 International Symposium on*, pages 1–4. IEEE, 2016.

[52] T. Oder, T. Schneider, T. Pöpppelmann, and T. Güneysu. Practical cca2-secure and masked ring-lwe implementation. *IACR Cryptology ePrint Archive*, 2016:1109, 2016. URL `http://eprint.iacr.org/2016/1109`.

[53] T. Oder, T. Schneider, T. Pöppelmann, and T. Güneysu. Practical cca2-secure and masked ring-lwe implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):142–174, 2018.

[54] P. J. J. O'Malley, R. Babbush, I. D. Kivlichan, J. Romero, J. R. McClean, R. Barends, J. Kelly, P. Roushan, A. Tranter, N. Ding, B. Campbell, Y. Chen, Z. Chen, B. Chiaro, A. Dunsworth, A. G. Fowler, E. Jeffrey, E. Lucero, A. Megrant, J. Y. Mutus, M. Neeley, C. Neill, C. Quintana, D. Sank, A. Vainsencher, J. Wenner, T. C. White, P. V. Coveney, P. J. Love, H. Neven, A. Aspuru-Guzik, and J. M. Martinis. Scalable quantum simulation of molecular energies. *Phys. Rev. X*, 6:031007, Jul 2016. doi: 10.1103/PhysRevX.6.031007. URL `http://link.aps.org/doi/10.1103/PhysRevX.6.031007`.

[55] E. Pednault, J. A. Gunnels, G. Nannicini, L. Horesh, T. Magerlein, E. Solomonik, and R. Wisnieff. Breaking the 49-qubit barrier in the simulation of quantum circuits. *arXiv preprint arXiv:1710.05867*, 2017.

[56] C. Peikert. Public-key cryptosystems from the worst-case shortest vector problem: Extended abstract. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 333–342, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-506-2. doi: 10.1145/1536414.1536461. URL `http://doi.acm.org/10.1145/1536414.1536461`.

[57] C. Peikert. *An Efficient and Parallel Gaussian Sampler for Lattices*, pages 80–97. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-14623-7. doi: 10.1007/978-3-642-14623-7_5. URL `http://dx.doi.org/10.1007/978-3-642-14623-7_5`.

[58] C. Peikert. Lattice cryptography for the internet. In *International Workshop on Post-Quantum Cryptography*, pages 197–219. Springer, 2014.

[59] C. Peikert, V. Vaikuntanathan, and B. Waters. *A Framework for Efficient and Composable Oblivious Transfer*, pages 554–571. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-85174-5. doi: 10.1007/978-3-540-85174-5_31. URL `http://dx.doi.org/10.1007/978-3-540-85174-5_31`.

[60] P. Peßl and M. Hutter. Pushing the limits of sha-3 hardware implementations to fit on rfid. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 126–141. Springer, 2013.

[61] P. Peßl and M. Hutter. Curved tags–a low-resource ecdsa implementation tailored for rfid. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 156–172. Springer, 2014.

[62] W. Pfaff, B. Hensen, H. Bernien, S. B. van Dam, M. S. Blok, T. H. Taminiau, M. J. Tiggelman, R. N. Schouten, M. Markham, D. J. Twitchen, and R. Hanson. Unconditional quantum teleportation between distant solid-state quantum bits. *Science*, 2014. ISSN 0036-8075. doi: 10.1126/science.1253512. URL `http://science.sciencemag.org/content/early/2014/05/28/science.1253512`.

[63] T. Pöppelmann and T. Güneysu. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. *LatinCrypt*, 7533:139–158, 2012.

[64] T. Pöppelmann and T. Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In *International Conference on Selected Areas in Cryptography*, pages 68–85. Springer, 2013.

[65] T. Pöppelmann and T. Güneysu. Area optimization of lightweight lattice-based encryption on reconfigurable hardware. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2796–2799, June 2014. doi: 10.1109/ISCAS.2014.6865754.

[66] T. Pöppelmann, L. Ducas, and T. Güneysu. Enhanced lattice-based signatures on reconfigurable hardware. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 353–370. Springer, 2014.

[67] R. Primas, P. Peßl, and S. Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 513–533. Springer, 2017.

[68] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 84–93, New York, NY, USA, 2005. ACM. ISBN 1-58113-960-8. doi: 10.1145/1060590.1060603. URL `http://doi.acm.org/10.1145/1060590.1060603`.

[69] O. Reparaz, S. Sinha Roy, F. Vercauteren, and I. Verbauwhede. *A Masked Ring-LWE Implementation*, pages 683–702. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. ISBN 978-3-662-48324-4. doi: 10.1007/978-3-662-48324-4_34. URL `http://dx.doi.org/10.1007/978-3-662-48324-4_34`.

[70] S. S. Roy, O. Reparaz, F. Vercauteren, and I. Verbauwhede. Compact and side channel secure discrete gaussian sampling. *IACR Cryptology ePrint Archive*, 2014:591, 2014.

[71] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede. *Compact Ring-LWE Cryptoprocessor*, pages 371–391. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-662-44709-3. doi: 10.1007/978-3-662-44709-3_21. URL `http://dx.doi.org/10.1007/978-3-662-44709-3_21`.

[72] C.-P. Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical computer science*, 53(2-3):201–224, 1987.

[73] M.-D. Shieh, J.-H. Chen, H.-H. Wu, and W.-C. Lin. A new modular exponentiation architecture for efficient design of rsa cryptosystem. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(9):1151–1161, 2008.

[74] P. W. Shor. Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Sci. Statist. Comput.*, 26:1484, 1997. doi: 10.1137/S0097539795293172.

[75] V. Singh. A practical key exchange for the internet using lattice cryptography. *IACR Cryptology ePrint Archive*, 2015:138, 2015. URL `http://eprint.iacr.org/2015/138`.

[76] V. Singh and A. Chopra. Even more practical key exchanges for the internet using lattice cryptography. *IACR Cryptology ePrint Archive*, 2015:1120, 2015. URL `http://eprint.iacr.org/2015/1120`.

[77] S. Sinha Roy, F. Vercauteren, and I. Verbauwhede. *High Precision Discrete Gaussian Sampling on FPGAs*, pages 383–401. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-662-43414-7. doi: 10.1007/978-3-662-43414-7_19. URL `http://dx.doi.org/10.1007/978-3-662-43414-7_19`.

[78] E. E. Targhi and D. Unruh. Post-quantum security of the fujisaki-okamoto and oaep transforms. In *Theory of Cryptography Conference*, pages 192–216. Springer, 2016.

[79] J. van de Pol and N. P. Smart. Estimating key sizes for high dimensional lattice-based systems. In *IMA International Conference on Cryptography and Coding*, pages 290–303. Springer, 2013.

[80] J. van Rantwijk. Pseudo random number generators as synthesizable vhdl code. `https://github.com/jorisvr/vhdl_prng/`, 2016.

[81] J. Wang and J. Bi. Lattice-based identity-based broadcast encryption scheme. *IACR Cryptology ePrint Archive*, 2010:288, 2010. URL `http://dblp.uni-trier.de/db/journals/iacr/iacr2010.html#WangB10`.

[82] E. Wenger and M. Hutter. Exploring the design space of prime field vs. binary field ecc-hardware implementations. In *Nordic Conference on Secure IT Systems*, pages 256–271. Springer, 2011.