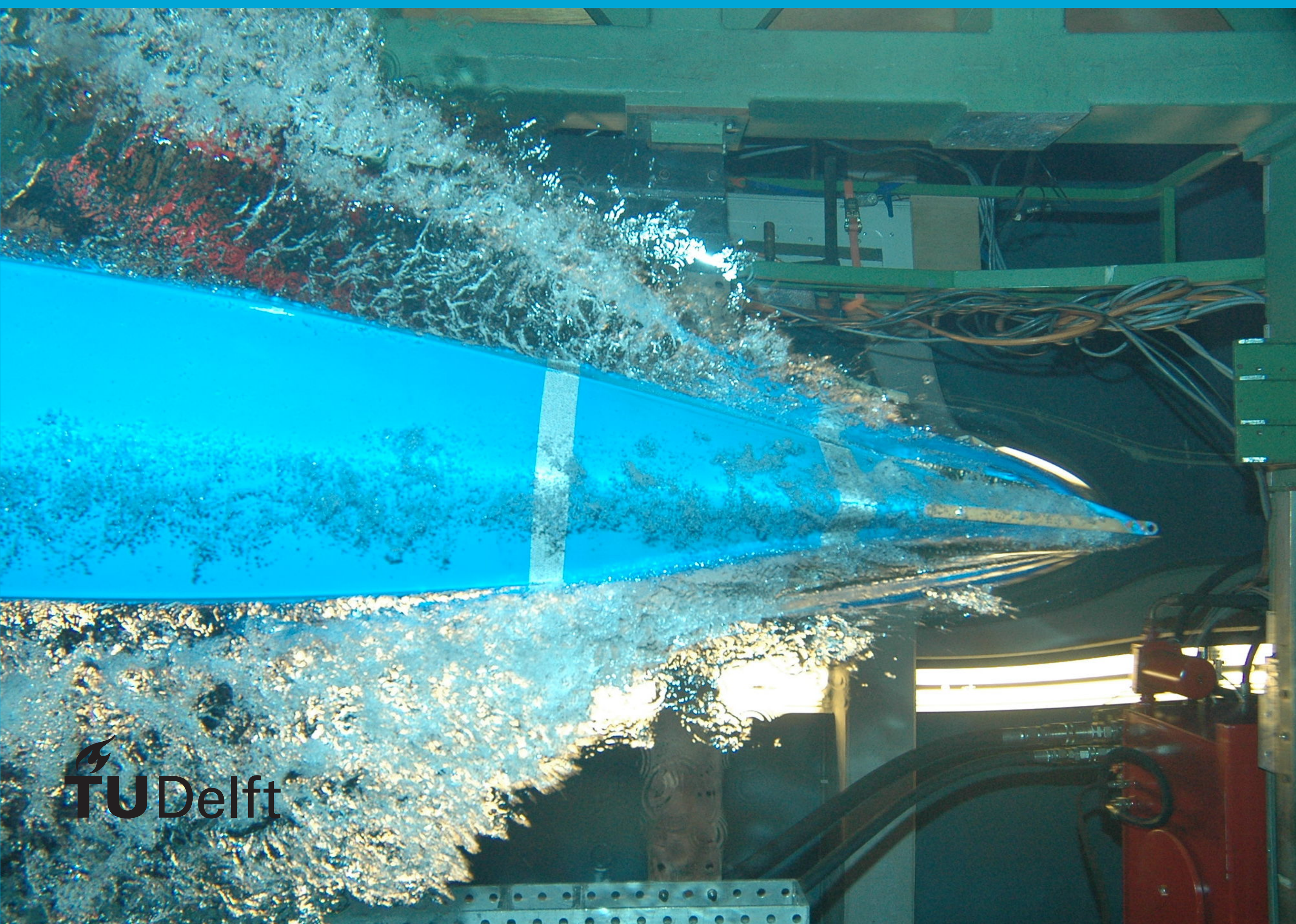
Hardware Implementation of the NTRU Deterministic Public Key Encryption

Erik Granneman



Hardware Implementation of the NTRU Deterministic Public Key Encryption

by

Erik Granneman

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended on September 20th, 2022.

Student number:	4227484	
Thesis number:	Q&CE-CE-MS-2022-02	
Project duration:	September 23, 2021 – September 20, 2022	
Thesis committee:	Dr. ir. J.S.S.M. Wong,	TU Delft, Chair
	Dr. ir. M. Taouil,	TU Delft, Supervisor, Proposer
	Dr. ir. T.G.R.M. Van Leuken	TU Delft, Jury member

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft Delft University of Technology

Acknowledgements

Throughout this project and the writing of this thesis I received support from many people, therefore I would like to take this opportunity to thank the people that have helped me during the past year.

First, I want to thank my supervisor Dr. Ir. M. Taouil for his guidance during the project, insight about all the research topics, and the feedback, discussion, and time at the many weekly progress meetings that we had throughout the year.

Secondly, I would like to thank Abdullah Aljuffri, for always being able to help me with my questions when needed, being there for the weekly progress meetings, and proofreading my thesis and slides, providing much appreciated feedback at all times.

Next, I want to thank Johanna Sepúlveda for helping me find this research topic and lending their insight about cryptography to guide me in the right direction.

Last, but not least, I would like to thank my friends and family for their emotional support, always being there for a chat, and to discuss discoveries throughout the project.

Many Thanks.

Contents

1 Introduction 5
- 1.1 Motivation 5
- 1.2 State-of-the-Art Post-Quantum Cryptography 7
- 1.3 Contribution 8
- 1.4 Thesis Outline 8

2 Post-Quantum Cryptography: An Overview 10
- 2.1 Introduction to the Post-Quantum Cryptography 10
- 2.2 Post-Quantum Scheme Overview 11
- 2.3 Code-based schemes 12
- 2.4 Isogeny-based schemes 15
  - 2.4.1 Diffie–Hellman key exchange 15
  - 2.4.2 Elliptic Curve Cryptography (ECC) 15
  - 2.4.3 Supersingular isogeny key exchange 16
- 2.5 Lattice-Based schemes 18
  - 2.5.1 The Shortest and Closest Vector Problem 18
  - 2.5.2 Learning with Error 19
  - 2.5.3 Scheme overview 22

3 The NTRU Cryptosystem 24
- 3.1 Introduction to NTRU Cryptosystem 24
- 3.2 Preliminaries 24
- 3.3 NTRU Parameters 25
- 3.4 The NTRU KEM Algorithm 27
  - 3.4.1 Keygen' 27
  - 3.4.2 Encapsulation 27
  - 3.4.3 Decapsulation 27
- 3.5 The NTRU DPKE Algorithm 27
  - 3.5.1 Key Generation 28
  - 3.5.2 Encryption 28
  - 3.5.3 Decryption 28

4 Design and Implementation 30
- 4.1 Related work in the NTRU domain 30
- 4.2 System overview 31
- 4.3 Input State 32
  - 4.3.1 Control signals for Input State 33
  - 4.3.2 BRAM Structure 33
  - 4.3.3 BRAM Input Handler 33
- 4.4 Encryption 34
  - 4.4.1 Convolution Module for Encryption 35
  - 4.4.2 Polynomial Register 36
  - 4.4.3 Lift Operation 36
- 4.5 Decryption 40
  - 4.5.1 Ternary Register 41
  - 4.5.2 Convolution Module for Decryption 42
  - 4.5.3 Reduction 44
- 4.6 Output State 45
- 4.7 Full System Overview 45

5 Results and Analysis 47
5.1 Experimental setup 47
5.2 Area Overhead Results 48
5.3 Performance Results 49
5.3.1 Lift Function Results 49
5.3.2 Convolution Function Results 52
5.3.3 Additional Performance Results 53
5.4 Implementation Discussion. 54
5.5 Security Analysis 55

6 Conclusion 57
6.1 Summary 57
6.2 Future Work. 57

Bibliography 59

Abstract

The increasing advancements in quantum computing have led to an increasing danger for the cyberspace. The current cryptographic algorithms that are used to enable secure communication across insecure channels have the potential to be brute-forced by sufficiently powerful quantum computers, endangering the security of many electronic devices and protocols that use popular algorithms such as RSA. While it is not feasible currently, these advancements in quantum computing are accelerating rapidly and the impact this could have on the security of the cyberspace is too great, therefore countermeasures must be considered. To protect against this threat, the National Institute of Standards and Technology (NIST) has started an initiative to work towards standardizing quantum-resistant cryptoschemes before the advancements in quantum computing reach such a level. This has led to a great amount of collaboration by researchers to develop and analyze the security of these quantum-proof schemes over the past six years.

This thesis explores the various post-quantum cryptoschemes that are currently being considered, outlining their differences and the potential advantage of using each scheme. While all of the current submissions are required to have a software implementation to be part of the submission, this is not the case for a hardware implementation. Hardware implementations can have different vulnerabilities than software implementations and, due to this, having one or preferably multiple hardware implementations available for these schemes would greatly advance the security analysis that can be performed for these candidates. Therefore, this thesis describes the hardware implementation process of one such scheme, NTRU, one of the longest standing lattice-based schemes, since this danger of quantum computing is equally dangerous for the many hardware devices and chips that are used worldwide. It discusses the various design decisions that have been made during the implementation and presents all functions that have been implemented to perform the encryption and decryption step of the deterministic public key encryption (DPKE) algorithm of NTRU. This implementation combines work that has been done for the previous NTRU submissions and adds additional logic to support the new and adjusted parts of the current NTRU algorithm.

The results show a fully functional encryption and decryption functionality of the NTRU cryptoscheme where the full encryption function can be performed in 3038 clock cycles while still maintaining a considerably low area usage, showing a speedup of 16 when compared to an optimized software implementation. Aside from this result, this thesis also provides several potential adjustments to the hardware implementation that can be made to reduce the decryption time at the cost of additional area so that the hardware can be tuned depending on the desired specifications.

1
Introduction

This chapter introduces the topic and goals that are addressed in this thesis, furthermore it will cover the relevance and contribution of this topic. Section 1.1 will present the motivation for this project and address the relevance by giving an introduction to post-quantum cryptography, where it stems from, and what it entails. Section 1.2 reviews the state-of-the-art when it comes to Post-Quantum Cryptography and the hardware implementations of these schemes. Section 1.3 covers the contributions of this thesis. Section 1.4 presents the thesis outline, giving a brief overview of each chapter.

1.1. Motivation

The current popular public-key encryption systems are vital for the Internet security of today by providing digital signatures and data encryption to facilitate highly secure communication across insecure channels. Algorithms such as Rivest-Shamir-Adleman (RSA), elliptic curve cryptography (ECC), and Diffie-Hellman have become a core part of many standards and protocols that are indispensible in many electronic devices worldwide. These heavily relied upon algorithms are being threatened by the steady rise of quantum computing due to the fact that these quantum computers can solve certain computational problems much faster than their classical counterparts. The common computational problems which underpins the security of these aforementioned algorithms: factoring integers or the discrete logarithm problem, can be solved by quantum computers using Shor's algorithm [1], which leads to significant ramifications for electronic security and privacy.

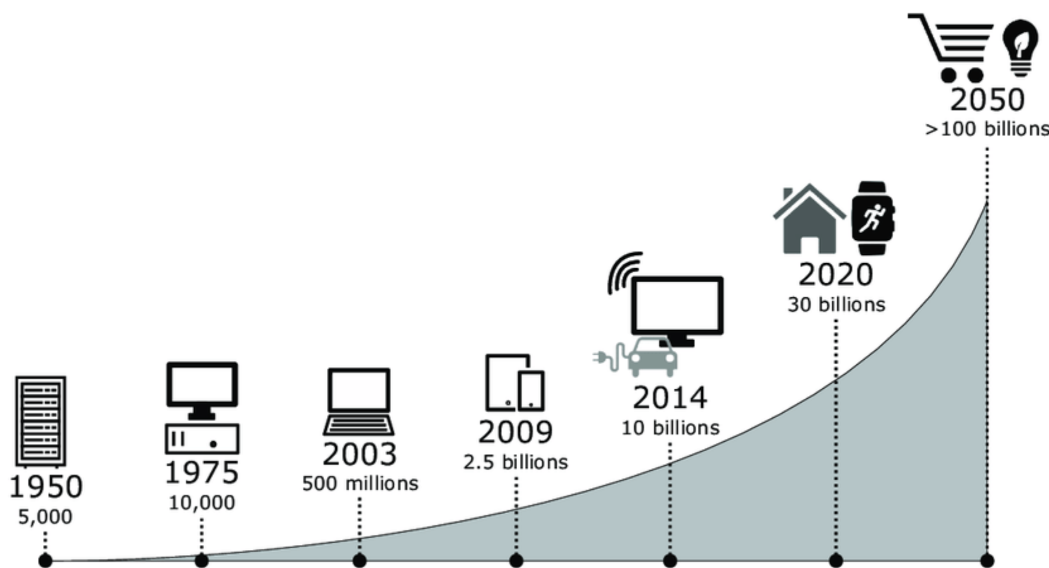


Figure 1.1: Expected adoption growth of IoT devices. [2]

It should come as no surprise that the number of public-key encryption systems is growing. This growth is being driven by the rise of internet-connected devices as well as the increase of online services. Consider, for example, the rapidly developing market for the Internet of Things (IoT). Over 100 billions IoT devices are expected to be linked to the internet by 2050, as shown in Figure 1.1, suggesting that this rapidly growing industry will see significant growth in the years to come. Therefore, it is necessary to maintain the security

level of public-key encryption systems at a satisfactory level in order to keep cyberspace safe and secure. The aforementioned quantum computing poses a serious danger to the current public-key encryption systems, and hence must be replaced.

Figure 1.2 shows an IBM roadmap of their quantum processor technology, showing the predicted increases and large growth of quantum computing over time. Even though quantum computing is still in its early stages and lacks the processing power to break these algorithms currently, this looming threat has led to an ever increasing interest in new algorithms that can be considered quantum-safe, or quantum-resistant. To share innovations, results, and findings about this, Post-Quantum Cryptography initiatives have been put in place such as the PQCrypto Conferences [3] and the NIST Post-Quantum Cryptography Standardization[4]. These initiatives have led to multitudes of quantum-resistant algorithms using many different approaches. Lattice-based cryptography is one of these approaches and one of the more leading candidates when it comes to Post-Quantum Cryptography with the largest amount of schemes in the current round of the NIST Post-Quantum Cryptography Standardization effort. In the current round almost half of the remaining schemes are lattice-based, indicating a great interest in these schemes specifically. As a requirement for these schemes that have been submitted there needs to be a full software implementation present, therefore, a hardware implementation is not always part of these submissions.

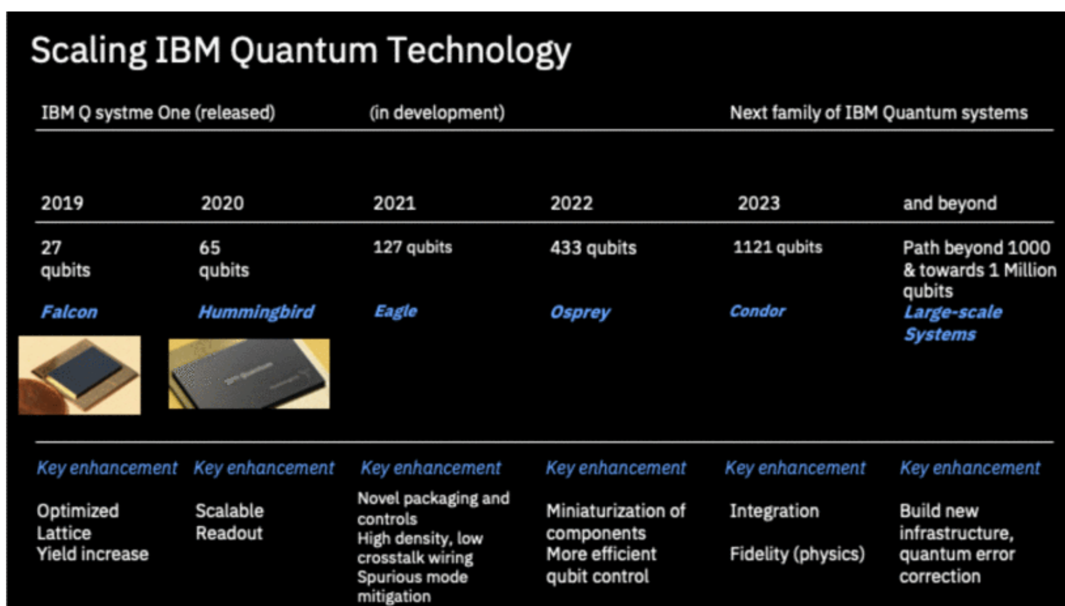


Figure 1.2: roadmap describing the scaling of IBM quantum processor technology in qubits and different engineering challenges that need to be tackled. [5]

Even though software implementations are desired, there is a growing demand for hardware implementations as well. The market for hardware-based encryption is anticipated to have a strong growth rate in the up coming years according to a recent report by Market Decipher [6], a market research and consultancy firm. With a compound annual growth rate (CAGR) of 29.3 percent between 2016 and 2022 as it is stated in the report, the whole market is expected to reach a value of USD 801.32 Billion by 2026, up from USD 413.85 Billion in 2022 [6] as shown in Figure 1.3. When compared with software-based systems, hardware-based encryption is superior in terms of both performance and energy efficiency, as it can provide a fast processing algorithm with low energy consumption. In addition to the advantages associated with efficiency, hardware-based cryptography also provides a robust solutions that are more secure. Hardware implementations, for example, are more protected against CPU side channel attacks such as spectre [7] and meltdown [8]. Other factors include protection against unauthorized code and tamper-proof or tamper-resistant key storage.

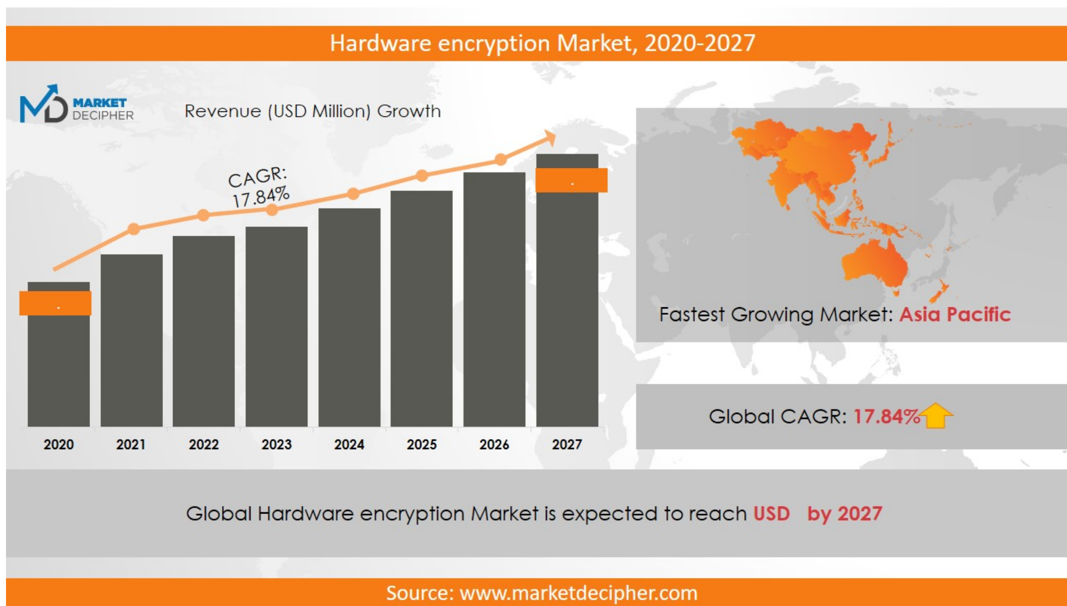


Figure 1.3: Expected growth of the global hardware encryption market. [6]

Due to the above reasons, this thesis focuses on the hardware implementation of a lattice-based cryptosystem called NTRU, which stands for N-th degree Truncated polynomial Ring Units. This cryptosystem is the only [9] lattice-based cryptosystem that has an error probability of zero and is therefore the most robust lattice-based cryptosystem, making it a great candidate for multitudes of different hardware.

1.2. State-of-the-Art Post-Quantum Cryptography

Stephen Wiesner [10] was the first to introduce quantum cryptography in 1968. This paper was originally rejected by IEEE Information Theory but eventually saw the light of print in 1983. He introduced conjugate coding, discussing a means of transmitting two messages so that either, but not both, can be received. In an example he uses the linear and circular polarization of light for this. Building upon this principle, Charles H. Bennett et al. [11] proposed a method of secure communications that was based on the conjugate observables introduced by Stephen Wiesner in 1992. This article discussed an experimental prototype which established the feasibility of this technology. Artur Ekert [12] developed a different method to distribute keys in 1991, based on quantum entanglement, which uses the physical phenomenon where the quantum state of particles cannot be described independently of another particle in the entangled group. Whereas previous work protected keys during transit, Ekert's approach also protects keys while they are stored. Additionally, because it is based on quantum entanglement, it can benefit from advanced quantum techniques like entanglement distillation [13][14], that were discovered years later. This led to prototypes of Ekert's approach following shortly after [15].

The introduction of Shor's algorithm by Peter Shor in 1994 [1] and arguably the start of the post-quantum era, lead to researchers not only looking into secure channels, but also into different cryptoschemes. This has led to researchers looking for algorithmic problems for which the resistance to quantum computer attacks is plausible. In 1996, Miklós Ajtai introduced the first lattice-based cryptographic construction [16], whose security was based around the hardness of well-studied lattice problems, the shortest vector problem [17] and short integer solutions [18]. In the same year another lattice-based cryptoscheme was introduced by Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman, NTRU [19]. This was the first cryptographic construction using polynomial rings.

Building upon the work of Ajtai [16] and research performed in the code-based field by R. J. McEliece [20], Goldreich, Goldwasser, and Halevi introduced the GGH public-key encryption scheme and signature scheme in 1997 [21]. The main idea behind GGH encryption and signatures is that the public key is considered a "bad" basis of some lattice, while the corresponding secret key is considered "good" basis of the same lattice. Even though both the GGH public-key encryption scheme and signature scheme had their security compromised [22] [23] later, the core ideas and structure of this scheme are still used to this day. Daniele Micciancio, took inspirating from NTRU and published work in 2002 [24] that took Ajtai's one-way/collision-resistant function and adapted it to work over polynomial rings, showing how this yields major efficiency improvements.

Additionally, in 2003, Oded Regev [25] gave several improvements to the original work of Ajtai in 1997. Some of his main contributions from this work are the introduction of gaussian measures and harmonic analysis over lattices. These techniques lead to a provably secure cryptoscheme under milder complexity assumptions than Ajtai's work, allowing for simpler algorithms.

Both Micciancio's and Regev's work are at the foundation of the currently considered post-quantum cryptosystems in the NIST Post-Quantum Cryptography Standardization effort. Kyber [26], one of the finalists, has its roots in the scheme of Regev, while both NTRU and NTRU prime have used Regev's work to improve upon the original NTRU cryptoscheme. [27][28]. Additionally, another finalist, Saber [29], and an alternative candidate FrodoKEM [30] refer back to these two works as well.

The above shows that the current lattice-based cryptoschemes have many similarities, yet all of the above schemes are considered a to be standardized. Even though their differences will be discussed in Section 2.5, this indicates that each cryptoscheme provides a unique benefit compared to the other ones to be considered and therefore further research into the hardware implementation of these schemes will always be beneficial for the future.

1.3. Contribution

In this thesis, we provide a hardware implementation for the NTRU cryptoscheme which is based on the NTRU version that has been submitted for the third round of the NIST Post-Quantum Cryptography Standardization effort. We analyzed steps of the encryption and decryption part of the algorithm, discussing the different approaches that could be taken in hardware and how these would affect the final implementation with regards to execution time, area, and complexity. After analysis we provide a low area implementation of the multiplication algorithm used in the algorithm. Lastly, we explored the implemented design and discuss possible vulnerabilities that are specific to hardware, such as side-channel analysis. The main contributions of the thesis are:

- **Research of current Post-Quantum schemes and their differences:** By analyzing all the different algorithm types that are currently part of the NIST Post-Quantum Cryptography Standardization effort and inspecting their functionalities we can provide an overview of the current Post-Quantum schemes and highlight the strengths and weaknesses of these schemes.
- **The comparison with state-of-the-art within the Post-Quantum domain:** We look at the current state-of-the-art at a high level, analyzing what has been done in the entire Post-Quantum domain. Additionally, this allows us to perform an in-depth analysis of the state-of-the-art when it comes to Lattice-based schemes, showing what research advancements have been made for each different Lattice-based scheme. Moreover, the analysis of the advancements that have been made towards NTRU hardware implementations specifically.
- **A low-cost hardware implementation:** We analyzed and validated hardware that can be implemented using lower area by performing polynomial multiplication without DSP units, the multiplication can be considered the most expensive operation of the NTRU algorithm. Additionally, we discuss additional options to tweak the area/speed tradeoff depending on hardware needs.
- **The succesful demonstration of a hardware implementation of the encryption and decryption blocks of NTRU:** We implemented, validated and evaluated all the required blocks to perform encryption and decryption according to the NTRU algorithm, discussing all important design decisions and providing full system diagrams and source code.

1.4. Thesis Outline

This thesis has been organized into six chapters, the first three chapters go over the topics that have been researched to provide sufficient background related to the thesis topic. The fourth and fifth chapter go over the design choices, implementation, and results of the proposed hardware. This is all followed by the final chapter which concludes the thesis. Each chapter focuses on covering a specific part and this is the following:

- Chapter 2 goes over the NIST Post-Quantum Cryptography Standardization initiative, providing an analysis of the different categories of algorithms and an in-depth discussion on the lattice-based cryptography and cryptosystems that are part of this category.

- Chapter 3 discusses the algorithm of the NTRU cryptosystem and looks into the state-of-the-art developments that have been made regarding NTRU implementations.
- Chapter 4 explains the hardware implementation that has been designed and goes over the different design decisions that have been made during the project.
- Chapter 5 discusses the implementation results and analyses these by discussing the impact that the design decisions that have been made have on the results.
- Chapter 6 concludes the thesis by providing a summary and discussing future work.

2
Post-Quantum Cryptography: An Overview

Cryptographic algorithms that are thought to be secure against a cryptanalytic attack by a quantum computer come in multiple different forms, this chapter will outline the different types of quantum-resistant cryptographic schemes that are candidates of the NIST Post-Quantum Cryptography Standardization program and give an overview of the lattice-based problems and their relevance in this project. Section 2.1 will introduce the Post-Quantum Cryptography, including the NIST standardization program. Section 2.2 will provide an overview of the different schemes that have been part of this program and how they can be categorized depending on which complex problem they are based on. Section 2.3 will discuss the code-based cryptosystem candidates and Section 2.4 will discuss the isogeny-based cryptosystem candidates. Finally, Section 2.5 will give a more in depth explanation of the approach used in this thesis, Lattice-based cryptography, explaining the various lattice problems before discussing the different cryptoschemes in this category and highlighting the differences between the different Lattice-based candidates.

2.1. Introduction to the Post-Quantum Cryptography

The term "post-quantum cryptography" refers to the development of encryption algorithms that are believed to be capable of defending against attacks associated with quantum computers. In the past, The concept of quantum computing was mostly theoretical. Nowadays after the development of actual computing units such as the Google's Sycamore quantum processor [31], it has become more and more applicable. When the day comes, when quantum computers with sufficient processing capacity are finally created, known algorithms such as RSA[32], ECC[33][34], and Diffie-Hellman[35] that are currently in use become completely useless. Even though it is unknown when a quantum computer that is good enough to brute force these algorithms will be devised, current encrypted data that holds senstive information can easily be stored and decrypted in the future. In addition, updating the infrastructure of the cryptosystem may take several years, which would delay the implementation of new cryptoschemes even further. As a consequence of this, these systems have an immediate and pressing need to find a solution to the problem presented by quantum computing. The National Institute of Standards and Technology (NIST) is aware of this fact, and in 2016 it launched a standardization program on post-quantum cryptography. The purpose of this program is to collect proposals and submissions in an effort to develop new cryptoschemes that are resistant to the effects of quantum computing. When the program is complete, the new standard algorithms will be revealed, and these cryptoschemes will be able to take the place of the public-key encryption systems that are now in use.

Because of the high level of mathematical complexity involved, the process of designing and assessing such algorithms will need a number of years and an iterative approach. This program was designed by the NIST in such a manner that it is capable of managing the process of collecting, testing, and ultimately proposing algorithms that are thought to be more resistant to quantum attacks. This procedure is carried out in a number of rounds [4], as described below:

- Round 1 - December 21, 2017 - 82 submissions received and 69 submissions accepted
- Round 2 - January 30, 2019 - 26 candidates moving on to the second round
- Round 3 - July 22, 2020 - 7 finalists announced with 8 alternative candidates

After announcing the candidates for each round, an evaluation phase would start where NIST performed internal review of the candidate algorithms. However, since this is a public project, the public and crypto community was invited to analyze the algorithms as well to determine which candidates were most promising. By taking this approach, inviting everyone to find problems and vulnerabilities with the candidate algorithms, they were eventually reduced to the 7 finalists and 8 alternative candidates that are in the current round 3 evaluation phase that has yet to finish.

In their most recent presentation[36], the NIST PQC Standardization states they intend to finish round 3 and announce candidates for a fourth round that will take up another 18-24 months, with a first set of new standards ready by 2024. Additionally, it is discussed that cryptanalytic results in round 3 have already shown security concerns for several of the signature algorithms (the ones based on the multivariate approach), which prompted NIST to plan another call for signature algorithms in the future.

2.2. Post-Quantum Scheme Overview

The primary objective of post-quantum cryptography is to find a quantum attacks-resistant algorithm that can be used to replace the currently used public key algorithms which are used in various sensitive applications. There are two main publications for public key algorithms namely key exchange and digital signature. Key exchange refers to the transmitting of the symmetric key (i.e., shared secret key) which is commonly used for the encryption and decryption of data during the communication between two parties. Due to the fact, that the key size of the symmetric algorithm is much smaller than the key size of public key algorithms, the symmetric key must be expanded using a technique that is mutually agreed upon by the people involved in the communication and is more difficult for an attacker to reverse. The Key Encapsulation Mechanism, abbreviated as KEM, is what's utilized in post-quantum cryptography as Key exchange method. KEM operates through the use of three primary functions 1) key generation: which generates a pair of public key and a private key, 2) Encapsulation: which takes as input a public key, and outputs a shared secret value and a ciphertext (i.e., encapsulation) of this secret value (i.e., the symmetric key), and 3) decapsulation: which takes as input the encapsulation and the private key, and outputs the shared secret key.

A digital signature, on the other hand, is an electronic signature that is used to validate both the integrity and authenticity of a message. Digital signatures are used in, for example, emails and digital documents. The digital signature creates a virtual fingerprint that is unique for a person or device and can be used to identify the sender and protect information in this way. A digital signature works by generating a unique hash of a message, or even a document, and then encrypting it by using the sender's private key. The receiver can then generate a hash of the message as well and compare it to the decrypted hash that they received. If both hashes are identical then this means that the sender is authenticated and the message has not been modified. Because this is a trust-based system and the public and private keys have to be established, in many cases a certificate authority is used. This certificate authority is a trusted third party that can validate a person or entity's identity and generate (or provide) a public/private key pair on their behalf. Once validated the certificate authority can issue a digital certificate that is signed by the certificate authority, which can be then be used to verify a person or entity when requested. Similarly to key exchange, digital signature algorithms are based on public key cryptography that has been shown to be vulnerable to attacks from quantum computers. Therefore, both the key exchange and digital signature algorithms must be replaced with post-quantum cryptographic algorithms so that they can be used in a post-quantum era.

Numerous post-quantum cryptographic algorithms have been presented as potential replacements for the existing public-key methods in KEM and digital signature applications. Table 2.1 displays the ones that made it through to the third round of the NIST evaluation process.

	Finalists	Alternatives
KEMs / Encryption	Kyber	Bike
	NTRU	FrodoKEM
	SABER	HQC
	Classic McEliece	NTRUprime
		SIKE
Signature	Dilithium	GeMSS
	Falcon	Picnic
	Rainbow	SPHINCS+

Table 2.1: List of Round 3 Candidates of the NIST PQC Standardization

These candidates can also be divided into five categories: code-based, isogeny-based, multivariate-based, hash-based, and lattice-based, depending on which complex problem the algorithm is based on. Several signature-based schemes are in the multivariate category that is currently being analysed due to security concerns, as well as two hash-based ones, while the code-based and isogeny-based schemes are exclusive to

KEM algorithms. Table 2.2 shows all the candidates grouped by their complex problem instead.

Scheme name	Category	Scheme name	Category
Kyber	Lattice-based	Classic McEliece	Code-based
NTRU	Lattice-based	BIKE	Code-based
SABER	Lattice-based	HQC	Code-based
FrodoKEM	Lattice-based	SIKE	Isogeny-based
NTRUprime	Lattice-based	GeMSS	Multivariate-based
Dilithium	Lattice-based	Rainbow	Multivariate-based
Falcon	Lattice-based	Picnic	Hash-based
		SPHINCS+	Hash-based

Table 2.2: List of Round 3 Candidates of the NIST PQC Standardization ordered by the complex problem that they are based on.

The cryptosystem that is being implemented in this thesis, NTRU[19], is a lattice-based KEM algorithm and, as such, the lattice-based KEM algorithms and the mathematical problems that they are based on will be discussed more in-depth. However, all other KEM finalists and alternative candidates will be discussed in this section to give additional background on the different approaches that are being taken when it comes to post-quantum cryptography. Due to the fact that hash-based and multivariate-based schemes are exclusive to signature schemes they will not be covered in this thesis.

2.3. Code-based schemes

At its core, coding theory is based around transmitting a message over a noisy channel. Given incoming message **m** encoded to linear code **M**, error during transmission **e**, and received linear code **C** the effect of such a noisy channel can be defined as:

$$\mathbf{C} = \mathbf{M} + e$$

where the addition can be considered a coordinate-wise addition. This received linear code **C** can be decoded back to message **m**, removing the error introduced by the noisy channel, provided that a proper encoding and decoding algorithm is used. A visual overview of this can be seen in figure 2.1, where **m'** indicates a potentially retrieved message **m**, depending on the algorithm that is used.

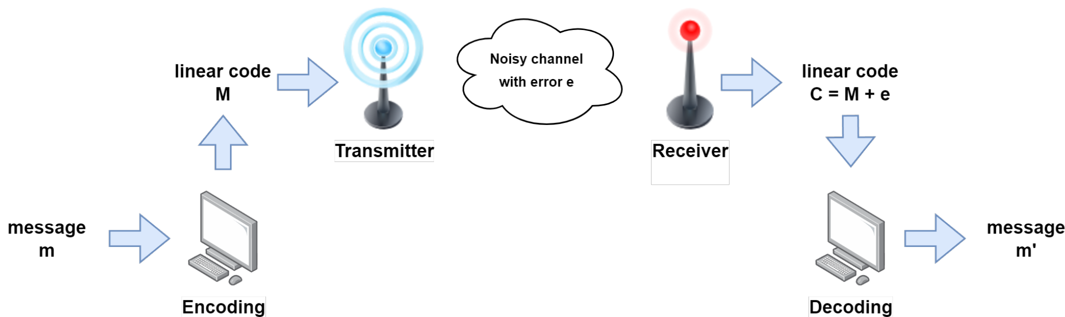


Figure 2.1: Fundamental structure of Coding Theory - Transmission over a Noisy channel.

Coding theory and cryptography seem to share a contradictory relationship, where in coding theory the purpose is to correct errors and retrieve the original message and in cryptography the purpose would be to obfuscate a message. The overlap lies in the use of special types of codes with an efficient decoding property so that an adversary has to distinguish such a code from a randomly generated code. By knowing the efficient decoding property and the information required to perform it, which would be considered the private key, this can function as a trapdoor function in a cryptosystem.

The three code-based schemes that are in round 3 of NIST PQC Standardization are:

- Classic McEliece[37] - KEM Finalist
- Bike[38] - KEM Alternative
- HQC[39] - KEM Alternative

These algorithms are all based on the hardness of decoding a general linear code, since this is known to be NP-Hard. NP-Hard stands for non-deterministic polynomial-time hardness and refers to the fact that these problems can not be solved in polynomial time, but it has been shown that the problem can eventually be solved. To analyze the similarities and differences between these three schemes, the McEliece cryptosystem will be shown below:

- Turn message **m** into a linear code **M** that has an efficient decoding algorithm **A**.
- Use linear code **M** to decide on a generator matrix **G** matching the family of codes that is used, where **G** is the private key and should be kept secret.
- Multiply generator matrix **G** by invertible matrix **S** and a permutation matrix **P** to get the public key **G'**.
- Add random noise **e** resulting in the ciphertext $\mathbf{C} = \mathbf{M} * \mathbf{G'} + \mathbf{e}$
- This ciphertext can not be decoded by an adversary, however, when someone has both the public key **G'** and private key **G**, the ciphertext can be reduced to $\mathbf{C} = \mathbf{M} * \mathbf{G} + \mathbf{e}$, which has the efficient decoding algorithm **A**.

Even though the generator matrix G is code specific, matrix S can be any random invertible matrix with the property that there exists a matrix B and when these two matrices are multiplied it holds that $SB = BS = I_n$, where I_n is the nxn identity matrix. Matrix P can be any random permutation matrix, which is a matrix that has exactly one entry of 1 in each column and each row with entries of 0 elsewhere. For additional clarification, a numerical example of the McEliece cryptosystem is provided in Example 2.1.

Example 2.1: McEliece Cryptosystem

Encryption:

1. Message **m** is converted into a linear code **M** = [0 1 0 1] with efficient decoding algorithm **A** and with generator matrix $\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$

2. Randomly generate invertible matrix **S** and permutation matrix **P** and compute public $\mathbf{G'} = \mathbf{S} \cdot \mathbf{G} \cdot \mathbf{P}$

$$\mathbf{S} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \mathbf{P} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{G'} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

3. Calculate the ciphertext $\mathbf{c} = \mathbf{M} \cdot \mathbf{G'}$ = [1 0 1 1 0 1 0] and add random noise such as the 1-bit error **e** = [0 0 0 1 0 0 0], resulting in $\mathbf{C} = \mathbf{c} + \mathbf{e}$ = [1 0 1 0 0 1 0]

Decryption:

1. To perform decryption, matrix **S** and **P** need to be inverted, both of which are trivial to invert:

$$\mathbf{S}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \mathbf{P}^{-1} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

2. Received is ciphertext **C** = [1 0 1 0 0 1 0], which can be multiplied with $\mathbf{P}^{-1}$ to undo the multiplication with **P** performed during encryption: $\mathbf{y} = \mathbf{C} \cdot \mathbf{P}^{-1}$ = [0 0 0 0 1 1 1]

3. Using efficient decoding algorithm **A** for **G**, the added error and multiplication with **G** can be removed resulting in $\mathbf{A}(\mathbf{y}) = \mathbf{M'} = \mathbf{M} \cdot \mathbf{S}$ = [0 0 1 0]

4. Multiplying with **S'** then results in $\mathbf{M'} \cdot \mathbf{S}^{-1} = \mathbf{M}$ = [0 1 0 1], retrieving the original linear code.

It should be noted that the McEliece cryptosystem [20] is **not** the Classic McEliece scheme [37], since both the Classic McEliece scheme and Bike scheme [38] are based on the Niederreiter cryptosystem [40] instead, which is a variation of the McEliece cryptosystem. The main difference between these two cryptosystems is that McEliece is based on the general decoding problem and Niederreiter is based upon the syndrome decoding problem [40].

The syndrome of a code is a vector that characterises the specific error of the code. This means that the syndrome will be a zero vector if there is no error and a non-zero vector when some error occured. The syndrome can be computed by multiplying the encoded message with the transpose of a code-specific parity-check matrix which needs to be generated. Even though the Niederreiter cryptosystem is equivalent in security to the McEliece cryptosystem, the main advantage of using the syndrome is that is much smaller than the encoded message and therefore any computations performed on it can be done faster, resulting in a more efficient cryptosystem.

In Classic McEliece the codes with efficient decoding property that are used are Goppa codes [41], due to the properties of these Goppa codes the permutation matrix **P** and invertible matrix **S** are no longer required in the algorithm. The advantage of this is that the decoding can be performed much more efficient due to the fact that it requires simpler computations now. Unfortunately, a large downside which makes it difficult to use for internet protocols is the public key size, which ends up being over a million bytes.

The Bike scheme [38] aims to resolve this downside of large bandwidth by using moderate-density

parity-check (MDPC) codes [42] over Goppa codes, resulting in a public key size of only ≈ 5000 bytes, which is much lower than the aforementioned Classic McEliece. However, it has not managed to get full CCA-security.

HQC, while also based on the syndrome decoding problem, does not use the hidden structure of codes like the other two schemes. It uses public Reed-Muller [43][44] and Reed-Solomon (RMRS) [45] codes and a random code to which a random error vector is added, this error vector is the private key. This results in the problem of decoding a random code, which is proven to be NP-complete (similar to NP-hard, but the solution to the problem has been computed instead of just in theory), except for when the private key is known. HQC has a low encryption rate, but more efficient decoding than the Bike scheme, however, it has a higher bandwidth at about ≈ 7000 bytes [39] for the public key.

The main attack strategy against the one-way functions of these code-based schemes is information-set decoding (ISD), introduced by Prange[46] in 1962.

2.4. Isogeny-based schemes

Isogeny-based schemes are based around the supersingular isogeny (SSI) problem introduced by Jao and De Feo in 2011[47], making it a relatively new problem compared to other ones that are decades of research ahead. This is the main reason why Supersingular Isogeny Key Encapsulation (SIKE)[48] is the only Isogeny-based scheme that is part of the current round of the NIST PQC Standardization effort. At the core this scheme involves arithmetic operations of elliptic curves over finite fields, it is analogous to the Diffie–Hellman key exchange protocol, but is based on walks in a supersingular isogeny graph. Due to this it also preserves forward-secrecy, which is a feature that gives assurances that session keys will not be compromised even if long-term secrets used in the session key exchange are compromised. Therefore, In order to explain the SSI mathematical problem of the Isogeny-based algorithm, first the Diffie–Hellman key exchange and the Elliptic Curve cryptographic algorithm needs to be explained, which will be covered more in-depth in the next subsections.

2.4.1. Diffie–Hellman key exchange

Diffie-Hellman key exchange is one of the most important developments in public-key cryptography and it is still implemented in many of today's security protocol as it is a method for safely exchanging and developing keys over an insecure channel. Exchanging information becomes challenging when the sender and receiver never had a chance to securely share keys for encryption and decryption and Diffie-Hellman key exchange is a protocol that provides a solution for this. It does this by securely developing shared secrets between two people that can then be used to derive keys from. These keys can then be used to transmit information in a secure manner using different algorithms that rely on this initial key exchange. For Diffie-Hellman key exchange, when the two users Alice and Bob want to communicate they take the following steps:

1. Alice and Bob agree between them on a large prime number **p**, and a generator (or base) **g** (where $0 < g < p$). This is done publicly and both **p** and **g** can be seen by an adversary.

2. Alice will choose a secret integer **a**, which is considered her private key and then computes $\mathbf{g^a} \bmod \mathbf{p}$, which is considered her public key. Bob chooses the private key integer **b**, and computes $\mathbf{g^b} \bmod \mathbf{p}$ instead.

3. Bob knows the public key $\mathbf{g^a} \bmod \mathbf{p}$ from Alice and can use his private key **b** to compute $(\mathbf{g^a})^\mathbf{b} \bmod \mathbf{p}$ resulting in $\mathbf{g^{ab}} \bmod \mathbf{p}$. Alice can do the same using her private key and Bob's public key and now both users have the shared secret $\mathbf{g^{ab}} \bmod \mathbf{p}$ while an adversary does not.

Diffie-Hellman key exchange is considered secure because $\mathbf{g^{ab}} \bmod \mathbf{p}$ takes a very long time to compute by only using the public knowledge of **p**,**g**, $\mathbf{g^a} \bmod \mathbf{p}$, and $\mathbf{g^b} \bmod \mathbf{p}$. However, like discussed earlier, this is no longer the case for quantum computers. Nevertheless, the core idea of combining two secret keys to establish a shared secret that is easy to compute in one way and difficult to reverse can still be used.

2.4.2. Elliptic Curve Cryptography (ECC)

An elliptic curve is a set of points that satisfy a specific mathematical equation and an elliptic curve can be written multiple forms, however, the standard form is:

$$y^2 = x^3 + ax + b$$

This standard form is known as the Weierstrass equation [49]. A property of these elliptic curves is that they have group structure, which means that when two points on the elliptic curve are added, the result will always be a point on the elliptic curve as well. Since scalar multiplication can be considered a sequence of point additions, this group structure makes it so that the multiplication of scalars is commutative, meaning that they can be computed in any order and still end up with the same result. The difficult problem at the core of these schemes is that scalar multiplication is easy to compute, but reversing it is difficult. Given two points **Q** and **P** it is difficult to find scalar **k** (See Equation 2.1). This is known as the Elliptic Curve Discrete Logarithm problem.

$$\mathbf{Q} = k \cdot \mathbf{P} \tag{2.1}$$

ECC may be used with Diffie-Hellman, which takes advantage of the algebraic structure of elliptic curves, allowing implementations to achieve a similar level of security with a much smaller key size. For example, a 224-bit elliptic-curve key has the same level of security as a similar 2048-bit RSA key [50]. Similarly to the original Diffie-Hellman which has public parameters **p** and **g**, the ECC version has several public parameters as well:

- **modulo p:** The field that is used to have a finite elliptic curve.
- **coefficients a and b:** The two coefficients that determine the elliptic curve in the Weierstrass equation.
- **generator point G:** This generator point can be used to get n amount of points on the elliptic curve by multiplying it with a scalar. This is a cyclic subgroup, which means that after n amount of points, the next scalar multiplication will result in the first point in the subgroup.
- **order n:** Based on the generator point this is the number of points **n** in the subgroup.
- **cofactor h:** The number of points on the elliptic curve divided by the order **n**, where a cofactor of 1 is the highest and most desirable value, indicating that every point on the curve is in the cyclic group.

By having this information available to both parties (and adversaries), all required information to define the elliptic curve and a generator point for it are known. Based on this, Alice and Bob can establish a shared secret using the following steps:

1. Alice will choose a secret integer **a**, which is considered her private key and then computes $\mathbf{A} = \mathbf{a} \cdot \mathbf{G}$, which is considered her public key. Bob chooses the private key integer **b**, and computes $\mathbf{B} = \mathbf{b} \cdot \mathbf{G}$ as his public key instead.
2. Bob knows the public key **A** and multiplies this with his private key **b** to compute the point on the elliptic curve that is $\mathbf{a} \cdot \mathbf{b} \cdot \mathbf{G}$. Similarly, Alice knows the public key **B** and multiplies this with her private key **a** to compute the same point on the elliptic curve that is $\mathbf{b} \cdot \mathbf{a} \cdot \mathbf{G}$ due to the fact that scalar multiplication is commutative.

Similarly to the original Diffie-Hellman algorithm, Alice and Bob have established a shared secret that is easy to compute for them since it is scalar multiplication, but hard to reverse for an adversary.

2.4.3. Supersingular isogeny key exchange

Isonogy is defined the following in the SIKE documentation: "Let E_1 and E_2 be elliptic curves over a finite field F_q. An isogeny ϕ: $E_1 \rightarrow E_2$ is a non-constant rational map defined over F_q which is also a group homomorphism from $E_1(F_q)$ to $E_2(F_q)$. If such a map exists we say E_1 is isogenous to E_2, and two curves E_1 and E_2 over F_q are isogenous if and only if $\#E_1(F_q) = \#E_2(F_q)$."[48]

A visual representation of this mapping can be seen in Figure 2.2.

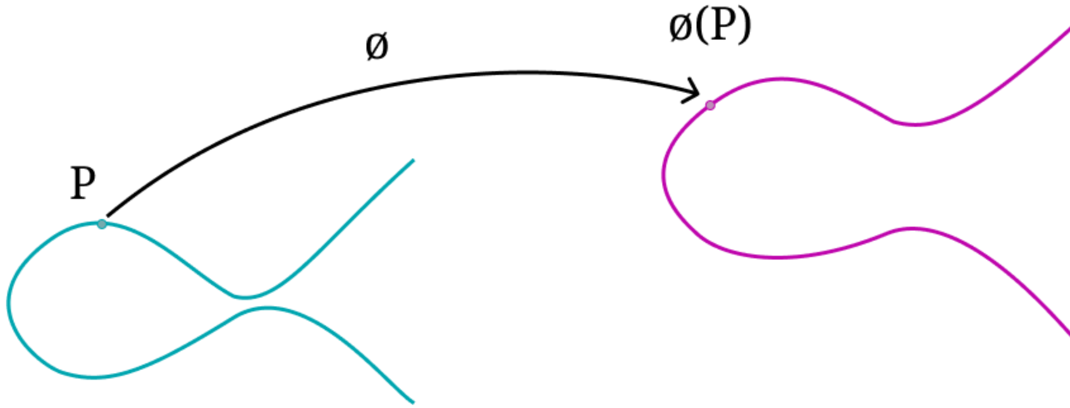


Figure 2.2: Visual representation of the mapping that is a function ϕ performed on a point P that determines whether an elliptic curve is isogenous. [51]

A difference should be made between isogeneous elliptic curves and isomorphic elliptic curves, since both of these definitions are used in elliptic curve theory. Elliptic curves are isomorphic if they have the same output (based on the Weierstrass equation) in:

$$j(E) = 1728\frac{4a^3}{4a^3 + 27b^2}$$

However, this does not make them isogeneous, since this is only the case when the aforementioned mapping exists for two elliptic curves. Both this mapping property between elliptic curves and the hardness of finding a scalar are at the core of the isogeny-based schemes.

In SIKE, the secret key is an octet string of integers taken from a key space that is determined by the security parameters of the scheme. This secret key determines the isogeny $\phi_{sk} : E \to E/H$. The public key is then based on the isogeny ϕ_{sk} and two predetermined points P and Q, which are public parameters. This public key is represented as a triplet of field elements representing the x-coordinates of three points under the isogeny:

$$\{x_{\phi_{sk}(P)}, x_{\phi_{sk}(Q)}, x_{\phi_{sk}(P-Q)}\}$$

A large disadvantage of SIKE, or Isogeny-based schemes in general, is still the fact that it is a relatively new problem. When it comes to confidence in a hard cryptographic algorithm the test of time has no substitute. However, it can also be seen as an advantage because the fact that this scheme has not suffered any security losses in the decade that it has existed should provide some confidence.

Additionally, Table 2.3 and 2.4 show the best known complexity for classical and quantum attacks respectively. It can be observed from these tables that there is barely an advantage to quantum computing when it comes to solving this problem, which means that the rise of quantum computing would have no effect on the security metrics of this scheme. The case where quantum computing is a slightly better option is when the available memory is 2^{96} which is a massive requirement and much more memory that computers have available currently.

	Best Classical 2^{96}	Best Classical 2^{64}	Best Classical 2^{40}
SIKEp434	117	133	135
SIKEp503	142	158	160
SIKEp610	183	199	201
SIKEp751	235	251	253

Table 2.3: The best known classical attack complexities (rounded base-2 logarithms) for the four SIKE instances.[52]

	Best Quantum 2^{96}	Best Quantum 2^{64}	Best Quantum 2^{40}
SIKEp434	124	147	178
SIKEp503	134	179	234
SIKEp610	181	189	307
SIKEp751	219	274	345

Table 2.4: The best known quantum attack complexities (rounded base-2 logarithms) for the four SIKE instances.[52]

2.5. Lattice-Based schemes

Lattices can be defined as the set of all integer linear combinations of basis vectors $\{b_1, b_2, ..., b_n\}$. In other words, a lattice would be all the points that can be reached by combining or scaling a selection of vectors. Lattices can appear in many different shapes based on which basis vectors are chosen. For example, $\mathbb{Z}^n$ is a lattice that is generated by standard basis of $\mathbb{R}^n$, where the standard basis is the set of vectors whose components are all zeroes, except one that equals 1. It should be noted that the scaling of vectors can only be performed using whole integers.

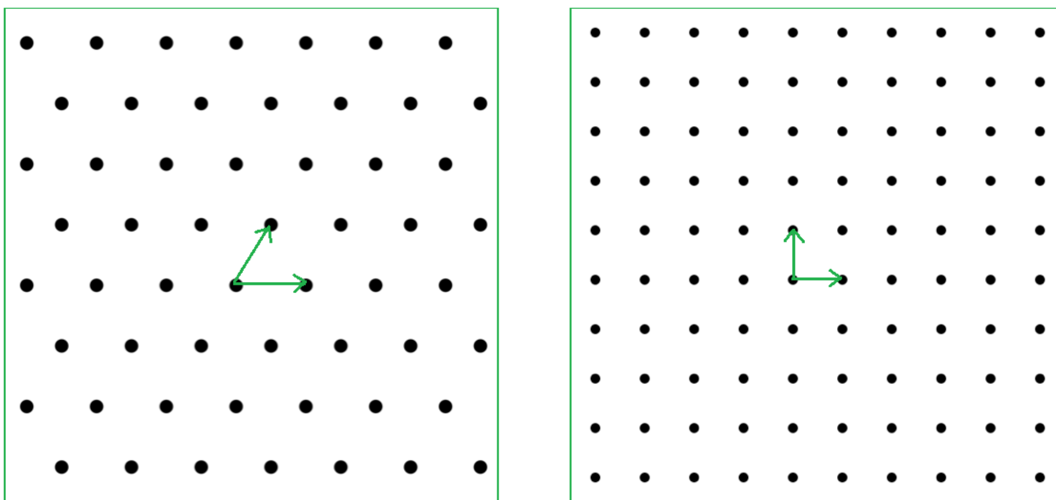


Figure 2.3: Example Lattices

Figure 2.3 shows two examples of a lattice with its basis vectors, note that these lattices would end up missing a lot of their points if these vectors were longer since vectors can only be scaled using integers and not with fractions, leading to many different kinds of lattices. An important property of these lattices is that any given lattice does not have just one basis. For example, the example lattice on the right in Figure 2.3 could also be generated by choosing the basis {(-1,0),(0,-1)} or {(2,1),(1,1)}.

Similarly to code-based and isogeny-based, lattice-based schemes are built around several complex problems that make it difficult for an adversary to get private information. There are several versions of this problem when it comes to lattices such as Learning with Error and Learning with Rounding, however, all of the currently used complex problems have a relation with either the Shortest Vector Problem or the Closest Vector Problem. Therefore, to discuss the more commonly used complex problems in the current lattice-based cryptoschemes, these two problems will be explained first.

2.5.1. The Shortest and Closest Vector Problem

Even though most of the current lattice-based cryptoschemes are not about solving the Shortest Vector Problem (SVP) [17], the problem remains at the core of these schemes and serves as a starting point to understand which complex problem each of these schemes is based on. The goal of the SVP is for an attacker to find the shortest vector from the origin when given the basis of a lattice. The zero vector is excluded from this problem and is considered a trivial answer.

The SVP can be considered special case of the Closest Vector Problem (CVP) [17], where the goal is to find the shortest vector from a given point to a point on the lattice. The SVP can be considered a CVP with the

given point being the origin. The SVP problem has been shown to be an NP-hard problem under randomized reductions by Ajtai [53].

These problems are simple in their essence, however, they get increasingly complex when the size of lattice increases [54]. With lattice-based cryptoschemes being prevalent when it comes to post-quantum schemes, much research has been performed around the complexity of these two problems to ensure lattice-based schemes are still secure. To illustrate the complexity, an SVP algorithm which is able to approximate the solution with a time complexity of $\mathcal{O}(N^3)$ was demonstrated in 2018 (Chuang, Fan, & Tseng, 2018)[55]. In 2021, a CPU and GPU based attack was presented by G. Falcao, F. Cabeleira, A. Mariano and L. Paulo Santos [54] giving an indication of the hardness of the Shorted Vector Problem and lattice-based cryptoschemes. This algorithm was tested on lattices of lower dimensions with the execution time for each dimension shown in Figure 2.4.

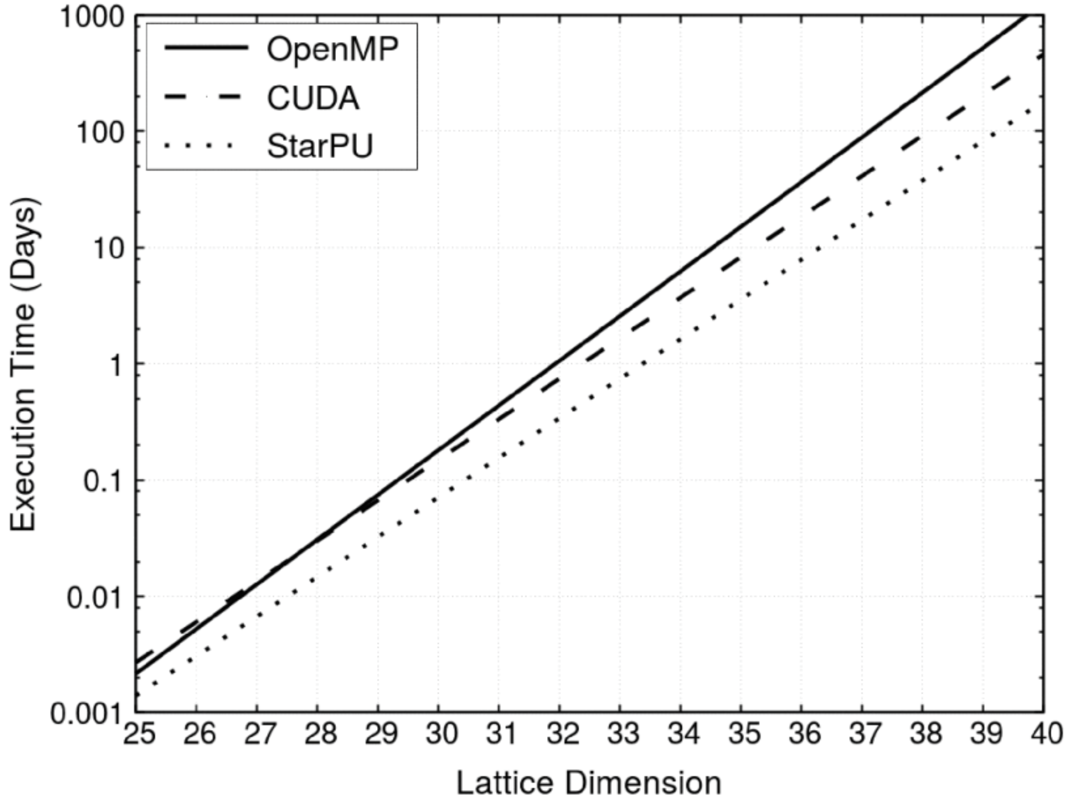


Figure 2.4: Execution times of Voronoi on a CPU, on a GPU and on the CPU+GPU platforms, based on curve fitting. [54]

2.5.2. Learning with Error

To discuss Learning with Error (LWE), which is the problem that is used by every lattice-based cryptographic KEM scheme except NTRU, a relation must first be drawn with the Shortest Vector Problem. These two problems are not directly related, however, both are related to the Small Integer Solutions (SIS) problem from Micciancio and Regev [18].

Algorithm 1 Small Integer Solutions (SIS)

1: Given a modulus q, a matrix $\mathbf{A}$ (mod q) and a $v < q$, find $\mathbf{y} \in \mathbb{Z}^n$ such that $\mathbf{Ay} \equiv 0 \pmod{q}$ and $\|\mathbf{y}\| \leq v$.

The SIS problem shown in Algorithm 1 is not exclusively a lattice problem but can be considered a lattice problem when looking at a modular lattice, or q-ary lattice, where q is an integer prime number [56]. In fact, in that case the SIS problem is to find the SVP for the lattice $\mathcal{L}_q^\perp(\mathbf{A})$ which is the lattice of matrix $\mathbf{A}$ where:

$$\mathcal{L}_q^\perp(\mathbf{A}) = \{\mathbf{z} \in \mathbb{Z}^n : \mathbf{Az} = 0 \text{ (mod q)}\}$$

This lattice $\mathcal{L}_q^\perp(\mathbf{A})$ can be seen as a linear translation of the lattice $\mathcal{L}$ modulo q. In a graphical form, the problem can be viewed like shown in Figure 2.5 by Chris Peikert [56].

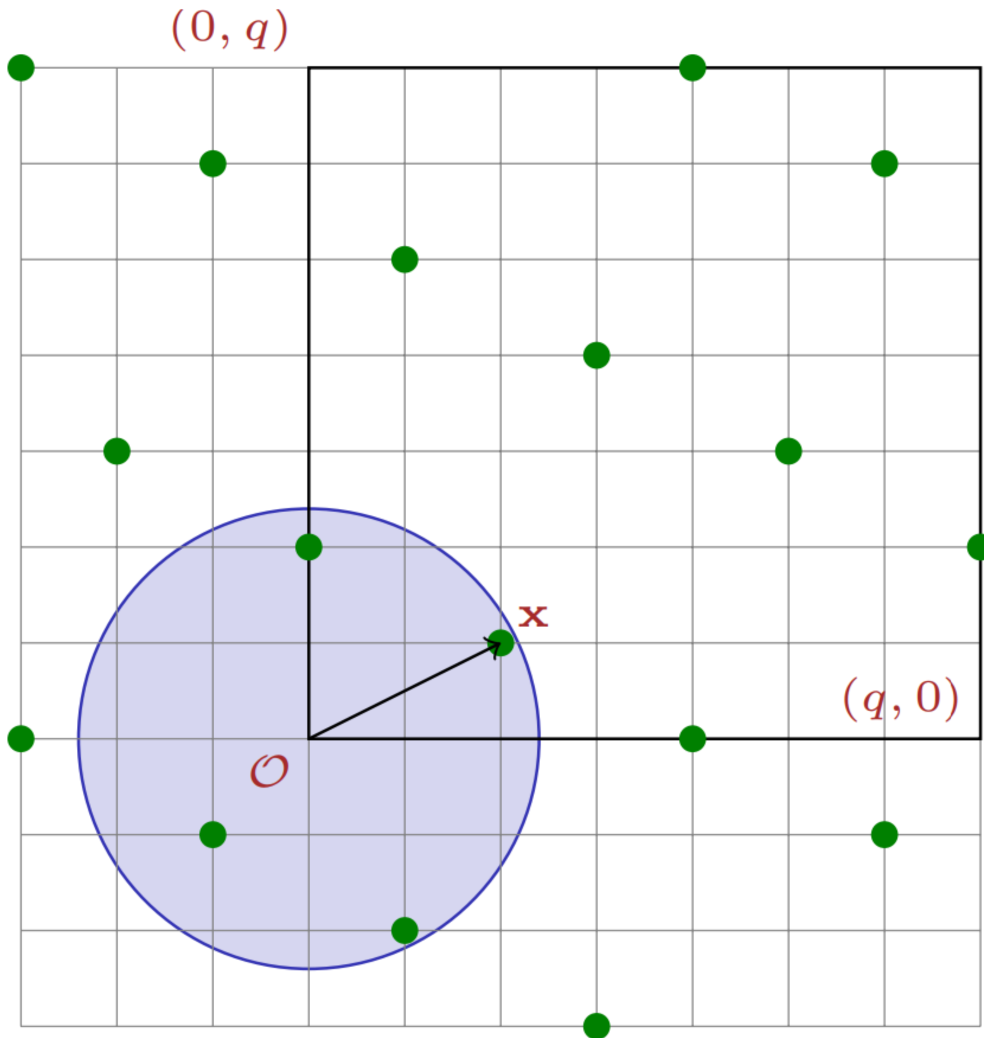


Figure 2.5: Illustration of the Small Integer Solution problem showing the modulo q window [56]

The modulo q window shown in Figure 2.5 is periodic since it ranged from 0 to q and an integer modulo q is 0. This holds for any translation and therefore also for the lattice $\mathcal{L}_q^{\perp}(\mathbf{A})$.

The LWE problem has a similar structure to this where there there is a polynomial **a** with coefficients sampled uniformly at random in $\mathbb{Z}_q^n$ where q is once again an integer prime number. The LWE problem is then to find the secret s when m samples ($m < n$) are given of the form (a, $a \cdot s + e$ mod q). In other words, finding the secret s, given many "noisy" inner products. Similarly to the SIS problem this can also be written in the case of a lattice to show the closer similarities between these different problems:

$$\mathcal{L}_q(\mathbf{A}) = \{\mathbf{z} \in \mathbb{Z}^n : \mathbf{z}^t \equiv s^t \mathbf{A} \text{ (mod q)}\}$$

The name of this problem stems from the fact that this would be a trivial problem were there no errors, since an adversary could solve for secret s when they know matrix **A** and are given the m samples, however, since there are small errors introduced these samples will not pinpoint a point on the lattice, but a point that is very close to it instead. The problem then becomes to find which point on the lattice is closest to the given point, resulting in a Closest Vector Problem as opposed to a Shortest Vector Problem in SIS.

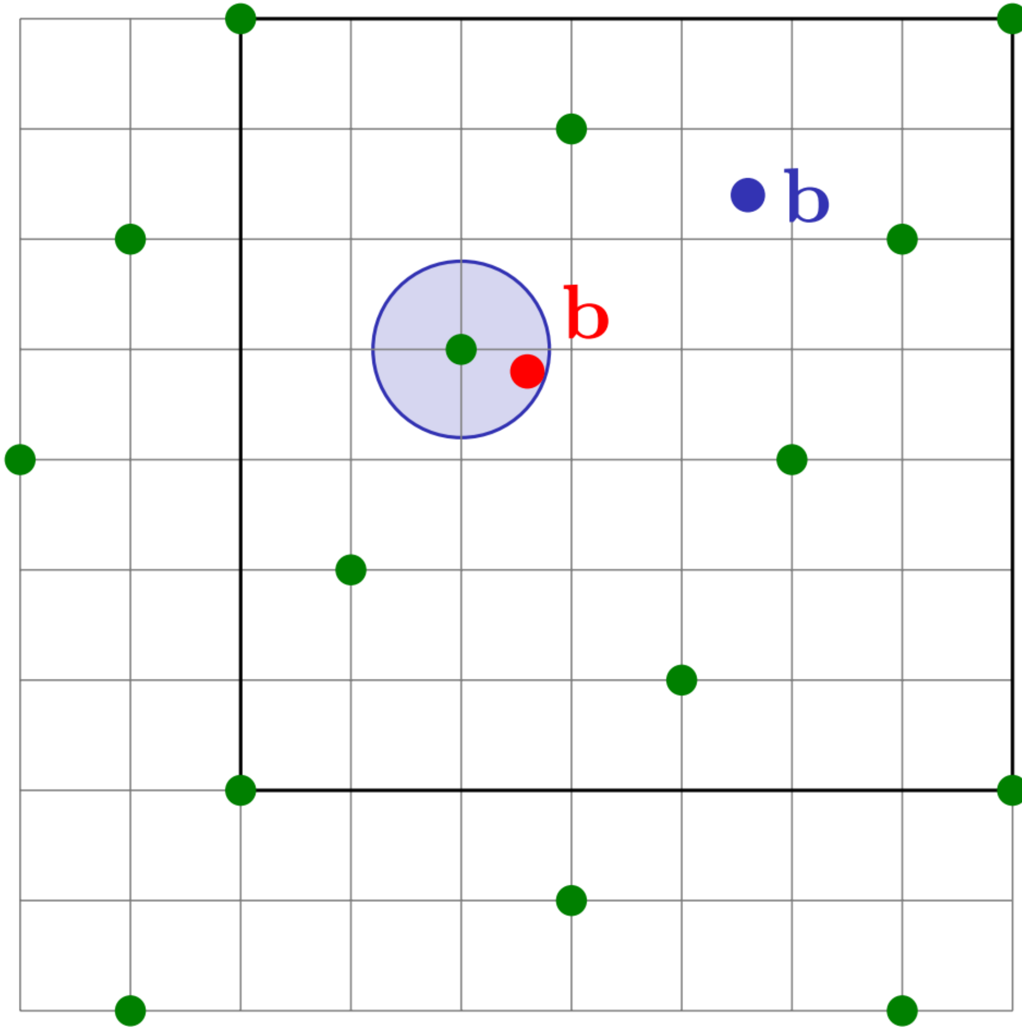


Figure 2.6: Illustration of the Learning with Error problem showing the modulo q window [56]

A graphical representation of the LWE problem can be seen in Figure 2.6, where the **b** in red is the given point that is close to an actual point on the lattice. Part of the hardness of this problem is that an attacker must distinguish the red **b** from the **b**, which is a randomly chosen point in the space $\mathbb{Z}_q^n$.

Two variations of the LWE problem are the Ring-LWE (RLWE) problem [57] and the Module-LWE (MLWE) problem [58]. The MLWE problem is a variation of the RLWE problem, which is a variation of the standard LWE problem. As the name suggests, the RLWE problem is the LWE problem that has been adapted to be used on a ring of polynomials. Instead of uniformly sampling polynomial **a** in $\mathbb{Z}_q^n$, **a** is sampled in R_q where:

$$R_q = \mathbb{Z}_q[x]/(x^n + 1)$$

R_q is the ring of polynomials where n is a power of two and q is an integer. This results in the RLWE problem where the objective is to find secret s when samples are given of the form $(a_i, a_i \cdot s + e_i \text{ mod q})$.

MLWE can be considered the "middle ground" problem that exists between the standard LWE problem and the RLWE problem. In essence, a module is a lattice of rank k over ring R, which means that for k=1 the MLWE problem is equivalent to the RLWE problem. the MLWE effectively takes the single ring elements **a** and **s** and replaces them with module elements over the same ring ($(R_q)^k$). Even though the cost of multiplication becomes higher in MLWE when compared to RLWE, the advantage of MLWE comes from the limitation on parameter n. One way to increase the security level is to increase the effective dimension of the ring that is being used and, for RLWE, this dimension is only dependant on parameter n. Therefore, when working with a ring that has a dimension of 1024, the dimension would have to be increased to 2048. In MLWE the dimension is equal to $n \cdot k$, which gives additional flexibility when increasing the dimension of a ring. A ring

with a dimension of 1024 (n=1024, k=1) could be increased to 1280 (n=256, k=5) or 1536 (n=512, k=3).

The final variation to be considered is the Learning with Rounding (LWR) problem which can be seen as a derandomization of LWE where the random noise e is replaced by a rounding modulo p < q. Rounding with this modulo p introduces a deterministic error instead of a random error which keeps the problem hard to solve. Polynomial **a** is still sampled uniformly at random in $\mathbb{Z}_q^n$ and samples are given of the form $(a, \lfloor a \cdot s \rceil_p$ mod q). This is because the random noise is removed by the rounding modulo p with high probability and therefore:

$$\lfloor a \cdot s + e \rceil_p = \lfloor a \cdot s \rceil_p$$

LWR has ring (RLWR) and module (MLWR) variants as well that have similar adjustments that the RLWE and MLWE version have when the random noise is replaced by the deterministic error introduced by LWR.

2.5.3. Scheme overview

A property that will be discussed more in depth in Chapter 3 is that NTRU is an outlier in the current selection of lattice-based schemes. It differs in that it is the only lattice-based scheme that does not use the LWE problem, which makes it a trapdoor one-way function. The other schemes are approximate commutative one-way functions and are therefore set up very different.

In NTRU f and its trapdoor f_{-1} are generated and can be considered the public key and private key respectively. The plaintext m can then be encrypted as c = f(m), while the ciphertext can be decrypted as m = f_{-1}(c), much like the cryptoschemes that are already in use.

Other schemes use the equation of the LWE problem to perform encryption and decryption by generating an approximate commutative function f_s and a random input **a**. The public key is then (a,b = f_s(a)) and the private key is s. The plaintext m is encrypted by using an approximate commutative one-way function g with a random element r so that c_1 = g_r(a) and c_2 = g_r(b) + E(m), where E is the encoding of an error-correction code. The ciphertext (c_1, c_2) can then be decrypted as m = D(c_2 - f_s(c1)), where D is the decoding of the generated error-correction code.

Although the other schemes are based on some variant of the LWE problem such as Ring-LWE or Module-LWE, FrodoKEM[30] is based around the "plain" LWE problem. This choice comes at the cost that it is a less computationally efficient algorithm when compared to the other lattice-based cryptoschemes, but allows FrodoKEM to be a prime choice when it comes to simplicity. The main reasoning for this choice is that the standard LWE problem has had much more security analysis performed compared to the alternatives, allowing more confidence when it comes to the security of this scheme. Furthermore, the added simplicity leaves a smaller potential for introducing additional vulnerabilities when creating a software or hardware implementation of this scheme. Current state-of-the-art research has not indicated any additional weaknesses when it comes to these LWE variants, however, further cryptoanalytic research on this topic could change this, which is one of the reasons why FrodoKEM is considered an alternative candidate despite its overall performance being lower than the other lattice-based candidates.[59]

Kyber[26] is based on the MLWE problem, making it one of the lattice-based schemes that takes advantage of the ring structure. The reason for this, and why FrodoKEM is considered less computationally efficient, is because these schemes can take advantage of the Number Theoretic Transform (NTT), which is a version of the discrete Fourier transform that has been specialized for quotient rings. Additionally, the Kyber cryptoscheme operations only consists of hash functions (Keccak variations), additions/multiplications in $\mathbb{Z}_q$, and the NTT, allowing the security level to be changed without having to perform drastic changes to the software or hardware implementation of the scheme. Kyber is one of the leading candidates and has already been integrated into libraries and systems in industry [60][61].

Closely related to Kyber is the Saber[29] cryptoscheme, which is based on the MLWR problem. A key difference is that in Saber the modulo parameters are all chosen to be a power of two so that explicit modular reduction is not required, additionally, this also simplifies the rounding operation that is required due to the MLWR problem. The trade-off of this parameter choice is that NTT-like polynomial multiplication is not natively supported for modulo that are a power of two, requiring Saber to use the classical multiplication algorithms. Normally this would be a large hit to the overall computational efficiency of the cryptoscheme, however, the differences between the MLWR and MLWE problem allow Saber to avoid performing full polynomial multiplications and instead perform circular shifts and additions.

Lastly, the two remaining schemes, NTRU [27] and NTRUprime [28], differ from the other schemes because they are based on the NTRU problem as opposed to the LWE problem. Even though they share similarities in name and the problem that they are based on, the fact that they are both still in the Nist PQC Standardization effort indicates that there must be major differences. The main distinction between NTRUprime and NTRU is

that NTRUprime abandons the use of the cyclotomic ring structure in favor of the field $\mathbb{Z}_q[\mathrm{x}]/(x_p - x - 1)$[62]. This unique choice of algebraic structure has led NTRUprime to be selected as an alternate candidate since new progress in the algabraic cryptoanalysis of cyclotomic structures could reduce the security confidence in the other lattice-based schemes whereas NTRUprime would be unaffected.

Alongside its longer history, lending more confidence towards its security, NTRU is based on a different security assumption than RLWE or MLWE, which has led it to be selected as a finalist alongside Kyber and Saber. The key generation of NTRU is slower than the other finalists due to the fact that key generation requires polynomial division in NTRU. Additionally, the parameter sets used by NTRU are not natively supported by NTT-like polynomial multiplication, resulting in lesser performance unless alternative solutions are found for these polynomial multiplications. Despite this, encryption in NTRU is considered to have a small performance advantage over other lattice-based scheme.

3
The NTRU Cryptosystem

NTRU was one of the first lattice-based cryptoschemes to be developed and therefore it has had several major iterations and adjustments over the years. This chapter will discuss the NTRU cryptosystem and the algorithm version that is part of the current NIST round 3 submission. Section 3.1 will introduce the NTRU cryptosystem and the historical changes that have happened to it. Section 3.2 discusses the various constants, notations, and formulas that are used throughout the NTRU documentation. Section 3.3 discusses the different parameters in the algorithm and how they differ, as well as which parameters are recommended. Section 3.4 will give an overview of the NTRU round 3 submission KEM algorithm and its functions. Similarly, section 3.5 discusses the NTRU round 3 submission DPKE algorithm and its functions. Additionally, this section covers how the different steps of the algorithm can be written in a more formal way that is suitable for hardware.

3.1. Introduction to NTRU Cryptosystem

With the first version of NTRU developed in 1996 by mathematicians Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman [19] [27], the NTRU cryptosystem has a long history of development. While originally described as a partially correct probabilistic public key encryption scheme (PPKE), it has been shown [19, section 4.2] NTRU can be made into a perfectly correct and deterministic public key encryption scheme (DPKE) by applying several transformations to the PPKE.

Even though most instantations of NTRU in literature have been based on the PPKE version [63] [64] [65] and with NTRU originally being split up into two separate PPKE schemes (NTRUEncrypt PPKE [27] and NTRU-HRSS-KEM PPKE [66]), the round 3 NIST PQC Standardization Process submission is a combined version of these two PPKE schemes and has applied the previously mentioned transformations to turn NTRU into a DPKE scheme. This DPKE scheme is based on the Saito-Xagawa-Yamakawa variant [67] of NTRU-HRSS-KEM and applies additional small changes [66] such as fixing parameter q as a function of parameter n in the algorithm.

The NTRU algorithm can be split up into two separate sections that will be discussed in order:

- **NTRU parameter set:** The NTRU algorithm has several subsets of parameters that not only alter the security level but also change several key functions that are used in the cryptoscheme.
- **NTRU KEM and DPKE:** The description of the Key Generation, Encryption, Decryption, Encapsulation, and Decapsulation routines that are used to go from a plaintext to a ciphertext and vice-versa.

Furthermore, the NTRU KEM and NTRU DPKE are discussed separately.

3.2. Preliminaries

Outside of the parameter sets and the algorithm the NTRU cryptoscheme use a set of definitions to describe the quotient rings that are used. These definitions are with respect to a fixed odd prime n where $(\mathbb{Z}/n)^x$ is the multiplicative group of integers modulo n and are the following [68]:

- $\boldsymbol{\Phi_1}$ is the polynomial $(x-1)$.
- $\boldsymbol{\Phi_n}$ is the polynomial $(x^n-1)/(x-1)$ which is equal to $x^{n-1}+x^{n-2}+...+1$.
- R is the quotient ring $\mathbb{Z}[x]/(\boldsymbol{\Phi_1\Phi_n})$.
- S is the quotient ring $\mathbb{Z}[x]/(\boldsymbol{\Phi_n})$.
- $R/3$ is the quotient ring $\mathbb{Z}[x]/(3,\boldsymbol{\Phi_1\Phi_n})$, indicating an additional modulo 3 operation compared to R.

- R/q is the quotient ring $\mathbb{Z}[x]/(q, \mathbf{\Phi_1 \Phi_n})$, indicating an additional modulo q operation compared to R. The canonical R/q-representative of $\mathbf{a} \in \mathbb{Z}[x]$ is written as $\underline{\text{Rq}}(\mathbf{a})$ which is the unique polynomial $\mathbf{b} \in \mathbb{Z}[x]$ of degree at most $n-1$ with coefficients in $\{-q/2, -q/2+1, ..., q/2-1\}$ such that $\mathbf{a} \equiv \mathbf{b}$ (mod (q, $\mathbf{\Phi_1 \Phi_n}$)).
- $S/3$ is the quotient ring $\mathbb{Z}[x]/(3, \mathbf{\Phi_n})$. The canonical $S/3$-representative of $\mathbf{a} \in \mathbb{Z}[x]$ is written as $\underline{\text{S3}}(\mathbf{a})$ which is the unique polynomial $\mathbf{b} \in \mathbb{Z}[x]$ of degree at most $n-2$ with coefficients in $\{-1, 0, 1\}$ such that $\mathbf{a} \equiv \mathbf{b}$ (mod (3, $\mathbf{\Phi_n}$)).
- S/q is the quotient ring $\mathbb{Z}[x]/(q, \mathbf{\Phi_n})$. The canonical S/q-representative of $\mathbf{a} \in \mathbb{Z}[x]$ is written as $\underline{\text{Sq}}(\mathbf{a})$ which is the unique polynomial $\mathbf{b} \in \mathbb{Z}[x]$ of degree at most $n-2$ with coefficients in $\{-q/2, -q/2+1, ..., q/2-1\}$ such that $\mathbf{a} \equiv \mathbf{b}$ (mod (q, $\mathbf{\Phi_n}$)).
- A polynomial is considered ternary if it has coefficients in $\{-1, 0, 1\}$, for example $R/3$ and $S/3$.
- $\mathcal{T}$ is the set of non-zero ternary polynomials of degree at most $n-2$. Following this definition it can be considered the set of canonical $S/3$-representatives.
- $\mathcal{T}_+$ is the subset of $\mathcal{T}$ which only contains polynomials with the non-negative correlation property.

3.3. NTRU Parameters

The NTRU parameter set consists of (n, p, q, $\mathcal{L}_f$, $\mathcal{L}_g$, $\mathcal{L}_r$, $\mathcal{L}_m$, Lift). In this set n, p, and q are coprime positive integers; $\mathcal{L}_f$, $\mathcal{L}_g$, $\mathcal{L}_r$, and $\mathcal{L}_m$ are sets of integer polynomials; and Lift is a function that injects $\mathcal{L}_m \to \mathbb{Z}$[x]. This parameter set is considered correct when:

$$(p \cdot \mathbf{r} \cdot \mathbf{g} \cdot \mathbf{f} \cdot \text{Lift}(\mathbf{m})) \bmod (\mathbf{\Phi_1 \Phi_n})$$

has coefficients in $\{-q/2, ..., q/2-1\}$ for all ($\mathbf{f}$, $\mathbf{g}$, $\mathbf{r}$, $\mathbf{m}$) $\in$ ($\mathcal{L}_f$, $\mathcal{L}_g$, $\mathcal{L}_r$, $\mathcal{L}_m$)[68].

Based on the definition of this parameter set there are two categories of parameter sets: NTRU-HPS and NTRU-HRSS. The NTRU-HPS parameters follow the previous NTRUEncrypt submission, while the NTRU-HRSS parameters follow the previous NTRU-HRSS-KEM submission. Following the documentation[68], the **NTRU-HPS** parameter set is an NTRU parameter set for which:

- n is a prime and both 2 and 3 are of order $n-1$ in $(\mathbb{Z}/n)^x$.
- p = 3.
- q is a power of two.
- $\mathcal{L}_f = \mathcal{T}$.
- $\mathcal{L}_g = \mathcal{T}(q/8-2)$.
- $\mathcal{L}_r = \mathcal{T}$.
- $\mathcal{L}_m = \mathcal{T}(q/8-2)$.
- Lift is the identity **m** mapped to **m**.

The **NTRU-HRSS** parameter set is an NTRU parameter set for which:

- n is a prime and both 2 and 3 are of order $n-1$ in $(\mathbb{Z}/n)^x$.
- p = 3.
- $q = 2^{\lceil 7/2+\log_2(n) \rfloor}$.
- $\mathcal{L}_f = \mathcal{T}_+$.
- $\mathcal{L}_g = \{\mathbf{\Phi_1} \cdot \mathbf{v} : \mathbf{v} \in \mathcal{T}_+\}$.
- $\mathcal{L}_r = \mathcal{T}$.
- $\mathcal{L}_m = \mathcal{T}$.

- Lift is the identity $\mathbf{m}$ mapped to $\mathbf{\Phi_1} \cdot \underline{S3}\ (\mathbf{m}/\mathbf{\Phi_1})$.

Even though both of these parameter sets are supported, the implementation will differ because of the differences in definition. For example, NTRU-HPS has a trivial Lift function but requires a more advanced sample function to get polynomials **r** and **m**.

Due to this difference the choice has been made to use an NTRU-HRSS parameter set because this will require the implementation of a lift function and this lift function can then optionally be removed so that the implementation works for an NTRU-HPS parameter set.

The NTRU documentation provides four recommended parameter sets, three of these follow the NTRU-HPS parameter set conditions and can be seen in Table 3.1. The remaining parameter set follows the NTRU-HRSS parameter set conditions and can be seen in Table 3.2.

Parameter	Set 1 (ntruhps2048509)	Set 2 (ntruhps2048677)	Set 3 (ntruhps4096821)
n	509	677	821
q	2048	2048	4096
Hash	SHA3_256	SHA3_256	SHA3_256
Sample_fixed_type_bits	15240	20280	24600
Sample_iid_bits	4096	5408	6560
Sample key bits	19304	25688	31160
Sample plaintext bits	19304	25688	31160
Packed ternary bytes	102	136	164
Packed poly bytes	699	930	1230
DPKE public key bytes	699	930	1230
DPKE private key bytes	903	1202	1558
DPKE plaintext bytes	204	272	328
DPKE ciphertext bytes	699	930	1230
KEM public key bytes	699	930	1230
KEM private key bytes	935	1234	1590
KEM ciphertext bytes	699	930	1230
KEM shared key bits	256	256	256

Table 3.1: Recommended parameters and derived constants for the NTRU-HPS parameter set. [68]

Parameter	Set 1 (ntruhrss701)
n	701
q	8192
Hash	SHA3_256
Sample_fixed_type_bits	n.a.
Sample_iid_bits	5600
Sample key bits	11200
Sample plaintext bits	11200
Packed ternary bytes	140
Packed poly bytes	1138
DPKE public key bytes	1138
DPKE private key bytes	1418
DPKE plaintext bytes	280
DPKE ciphertext bytes	1138
KEM public key bytes	1138
KEM private key bytes	1450
KEM ciphertext bytes	1138
KEM shared key bits	256

Table 3.2: Recommended parameters and derived constants for the NTRU-HRSS parameter set. [68]

Even though all the parameters shown in Table 3.1 and 3.2 have an impact on the performance, it can be seen that the chosen NTRU-HRSS set has the highest value of q. A high value of n or q increases the size of all

polynomials used in the algorithm and therefore this higher value of q will have a large impact on the overall performance. However, this does mean that swapping parameter sets should always improve the performance, making the implementation and results of using the NTRU-HRSS parameter set the worst-case scenario.

3.4. The NTRU KEM Algorithm

The NTRU KEM algorithm, where KEM stands for Key Encapsulation Mechanism, is the outer shell of the NTRU algorithm. It calls upon functions of the NTRU DPKE algorithm and is considered the only part that is exposed to the user. The NTRU DPKE algorithm requires the use of a complex padding mechanism to be considered CCA2-secure [66] and by applying the KEM algorithm this is no longer required, reducing the complexity of the algorithm. Additionally, key encapsulation mechanisms are the central building block in (authenticated) key exchange constructions, such as TLS [69].

KeyGen $(seed)$	Encapsulate $(\mathbf{h})$	Decapsulate $((\mathbf{f}, \mathbf{f}_p, \mathbf{h}_q, s), \mathbf{c})$
1. $((\mathbf{f}, \mathbf{f}_p, \mathbf{h}_q), \mathbf{h}) \leftarrow$ KeyGen$'(seed)$	1. $coins \leftarrow_\$ \{0,1\}^{256}$	1. $(\mathbf{r}, \mathbf{m}, fail) \leftarrow$ Decrypt$((\mathbf{f}, \mathbf{f}_p, \mathbf{h}_q), \mathbf{c})$
2. $s \leftarrow_\$ \{0,1\}^{256}$	2. $(\mathbf{r}, \mathbf{m}) \leftarrow$ Sample_rm$(coins)$	2. $k_1 \leftarrow H_1(\mathbf{r}, \mathbf{m})$
3. return $((\mathbf{f}, \mathbf{f}_p, \mathbf{h}_q, s), \mathbf{h})$	3. $\mathbf{c} \leftarrow$ Encrypt$(\mathbf{h}, (\mathbf{r}, \mathbf{m}))$	3. $k_2 \leftarrow H_2(s, \mathbf{c})$
	4. $k \leftarrow H_1(\mathbf{r}, \mathbf{m})$	4. if $fail = 0$ return k_1
	5. return $(\mathbf{c}, k)$	5. else return k_2

Figure 3.1: NTRU KEM

The NTRU KEM algorithm can be seen in Figure 3.1 and shows three functions: KeyGen', Encapsulate, and Decapsulate.

3.4.1. Keygen'

Even though most of the key generation is performed in the NTRU DPKE algorithm, the keygen' function has two purposes in the KEM algorithm:

- Give a random seed input to the DPKE key generation function
- append a 256-bit pseudo random function key called **s** to the private key that can later be used for validation.

Outside of these two functionalities the keygen' function serves as overhead for the actual keygen function in the DPKE algorithm and returns the public and private key to the encapsulation and decapsulation function.

3.4.2. Encapsulation

Before encryption can be performed, the input polynomials **r** and **m** need to be computed using a sample function in the encapsulation stage. This sample function is effectively a parameter since multiple different sample functions can be used, however, the choice of sample function can affect bandwidth and security. For the NTRU-HRSS parameter set the Sample_rm function provided in the documentation splits a byte array in half, resulting in two separate byte arrays for both **r** and **m** and then converts these byte arrays to ternary polynomials ($\mathcal{T}$) that can be used as input for the encryption function. Aside from sampling the input for the encryption function, the encapsulation step also performs a hash function on these inputs to generate a shared key which, alongside the returned ciphertext, are used in the decapsulation phase to ensure security.

3.4.3. Decapsulation

The decapsulation stage uses the output of the decryption stage to generate one of two possible outputs. When a fail state is returned by the decryption step it will return a random key, which is a hash function of s and **c**. However, when the decryption output is considered valid it will return the shared key that has also been generated in the encapsulation step.

3.5. The NTRU DPKE Algorithm

The NTRU DPKE algorithm can be considered the main part of the NTRU cryptosystem, this part of the algorithm securely encrypts a plaintext into a valid ciphertext and securely decrypts a ciphertext into a valid plaintext.

$\mathsf{KeyGen}'(seed)$	$\mathsf{Encrypt}(\mathbf{h},(\mathbf{r},\mathbf{m}))$	$\mathsf{Decrypt}((\mathbf{f},\mathbf{f}_p,\mathbf{h}_q),\mathbf{c})$
1. $(\mathbf{f},\mathbf{g}) \leftarrow \mathsf{Sample_fg}(seed)$	1. $\mathbf{m}' \leftarrow \mathsf{Lift}(\mathbf{m})$	1. if $\mathbf{c} \not\equiv 0 \pmod{(q,\mathbf{\Phi}_1)}$ return $(0,0,1)$
2. $\mathbf{f}_q \leftarrow (1/\mathbf{f}) \bmod (q,\mathbf{\Phi}_n)$	2. $\mathbf{c} \leftarrow (\mathbf{r}\cdot\mathbf{h}+\mathbf{m}') \bmod (q,\mathbf{\Phi}_1\mathbf{\Phi}_n)$	2. $\mathbf{a} \leftarrow (\mathbf{c}\cdot\mathbf{f}) \bmod (q,\mathbf{\Phi}_1\mathbf{\Phi}_n)$
3. $\mathbf{h} \leftarrow (3\cdot\mathbf{g}\cdot\mathbf{f}_q) \bmod (q,\mathbf{\Phi}_1\mathbf{\Phi}_n)$	3. return $\mathbf{c}$	3. $\mathbf{m} \leftarrow (\mathbf{a}\cdot\mathbf{f}_p) \bmod (3,\mathbf{\Phi}_n)$
4. $\mathbf{h}_q \leftarrow (1/\mathbf{h}) \bmod (q,\mathbf{\Phi}_n)$		4. $\mathbf{m}' \leftarrow \mathsf{Lift}(\mathbf{m})$
5. $\mathbf{f}_p \leftarrow (1/\mathbf{f}) \bmod (3,\mathbf{\Phi}_n)$		5. $\mathbf{r} \leftarrow ((\mathbf{c}-\mathbf{m}')\cdot\mathbf{h}_q) \bmod (q,\mathbf{\Phi}_n)$
6. return $((\mathbf{f},\mathbf{f}_p,\mathbf{h}_q),\mathbf{h})$		6. if $(\mathbf{r},\mathbf{m}) \in \mathcal{L}_r \times \mathcal{L}_m$ return $(\mathbf{r},\mathbf{m},0)$
		7. else return $(0,0,1)$

Figure 3.2: NTRU DPKE

The NTRU DPKE algorithm is shown in Figure 3.2 and shows the three functions that are part of it: KeyGen, Encrypt, and Decrypt.

3.5.1. Key Generation

Key Generation starts by computing the polynomials **f** and **g** using a sample function, similarly to the generation of **r** and **m** in encapsulation. Similarly, this sample function can be considered a parameter because there are multiple valid options available (that could have an effect on the bandwidth and security). The NTRU documentation provides a sample function, but it is not required to use this one specifically so long as the sample function output meets the parameter set requirements. For the NTRU-HRSS parameter set the Sample_fg function provided splits a byte array in half, resulting in two separate byte arrays for both **f** and **g** and then converts these byte arrays ternary polynomials that satisfy the non-negative correlation property ($\mathcal{T}_+$).

To compute the public key **h** and private key ($\mathbf{f},\mathbf{f_p},\mathbf{h_q}$) the key generation component has to compute three polynomial divisions and one polynomial multiplication. Additionally, two of these polynomial divisions are performed on the quotient ring $\mathbb{Z}[x]/(q, \mathbf{\Phi_n})$ while the other division and multiplication are performed on the quotient rings $\mathbb{Z}[x]/(3, \mathbf{\Phi_n})$ and $\mathbb{Z}[x]/(q, \mathbf{\Phi_1\Phi_n})$ respectively. After key generation, the secret **s** is appended to the private key by the KeyGen' function which is used for security checks in the decapsulation stage.

When it comes to hardware implementations, key generation is an expensive operation due to these divisions and multiplications to get the public and private key. Even though the polynomial multiplication is used in both the encryption and decryption stages, the division is exclusive to the key generation step and will therefore require additional separate hardware to compute these for any implementation.

3.5.2. Encryption

The encryption function consists of three operations:

1. The lift function, which is **m** mapped to $\mathbf{\Phi_1} \cdot \underline{S3}\ (\mathbf{m}/\mathbf{\Phi_1})$ for the NTRU-HRSS parameter set.
2. Polynomial multiplication of $\mathbf{r}\cdot\mathbf{h}$ on the quotient ring $\mathbb{Z}[x]/(q, \mathbf{\Phi_1\Phi_n})$
3. Polynomial addition of $(\mathbf{r}\cdot\mathbf{h})+\mathbf{m}'$, which is the lifted version of **m**.

An important property of this multiplication is that **r** is a ternary polynomial, which means that it can only have a value of {-1,0,1}. Which allows for simplifications of the polynomial multiplication that will be discussed in section 4.1.

Additionally, the final output $(\mathbf{r}\cdot\mathbf{h})+\mathbf{m}'$ is dependant on both the lift output and the polynomial multiplication result, while both of these functions are independant of eachother. For hardware implementation this allows both of these computations to be performed in parallel, making the encryption more efficient.

3.5.3. Decryption

The decryption function consists of seven operations:

1. A validity check on input **c** to ensure that $\mathbf{c} \equiv 0$ (mod $(q, \mathbf{\Phi_1})$. Otherwise output fail.
2. Polynomial multiplication of $\mathbf{c}\cdot\mathbf{f}$ on the quotient ring $\mathbb{Z}[x]/(q, \mathbf{\Phi_1\Phi_n})$
3. Polynomial multiplication of $\mathbf{a}\cdot\mathbf{f_p}$ on the quotient ring $\mathbb{Z}[x]/(3, \mathbf{\Phi_n})$
4. The lift function, which is **m** mapped to $\mathbf{\Phi_1} \cdot \underline{S3}\ (\mathbf{m}/\mathbf{\Phi_1})$ for the NTRU-HRSS parameter set.

5. Polynomial subtraction of $\mathbf{c} - \mathbf{m}'$
6. Polynomial multiplication of $(\mathbf{c} - \mathbf{m}') \cdot \mathbf{h_q}$ on the quotient ring $\mathbb{Z}[x]/(q, \boldsymbol{\Phi_n})$
7. A validity check on output $(\mathbf{r},\mathbf{m})$ to ensure that $\mathbf{r} \in \mathcal{L}_r$ and $\mathbf{m} \in \mathcal{L}_m$.

For two of the polynomial multiplications in decryption the same property that held in the encryption function holds, the multiplication has one ternary polynomial input. However, the final multiplication $(\mathbf{c} - \mathbf{m}') \cdot \mathbf{h_q}$ does not have this property. This last multiplication is far more expensive than any other operations in the NTRU DPKE due to the fact that two polynomials with n coefficients of size q are multiplied here. In the chosen NTRU-HRSS parameter set this results in polynomials with 701 coefficients of size 8192, requiring a vast amount of hardware to compute such a multiplication if traditional methods are used.

Furthermore, it can be seen that apart from step 1 and step 7, each step is dependent on the output of the previous step. This reduces the amount of potential parallelism that can be done and therefore puts additional pressure on the efficiency of the polynomial multiplication. All three polynomial multiplications are performed on a different quotient ring as well, which means that a different form of modular reduction has to be performed at the end of each of these multiplications.

4
Design and Implementation

This chapter will go over the entire hardware implementation process in a block-by-block manner. Section 4.1 looks at previous work related to NTRU hardware implementations and will go over the multiplication, which is the most expensive operation in the algorithm, and several methods to handle this efficiently. Section 4.2 will give a high-level overview of the entire system and the main states that it uses. Section 4.3 will discuss the input state, going over how communication with the outside works and how inputs are read. Additionally, this section will discuss how the Block RAM (BRAM) has been set up within the hardware architecture and what it does. Section 4.4 will then go over each functional block that is used in the encryption state, explaining the design decisions made around the overall system functions and how these are taken into account within these functional blocks. Section 4.5 discusses additions that are made to the architecture to implement the decryption step and discusses any adjustments made to the other blocks. Finally, section 4.6 explains how the outputs combined, stored, and handled while section 4.7 will give an overview of the full system where all the functional blocks and their details are combined to give a complete picture of the implementation.

4.1. Related work in the NTRU domain

Compared to the other lattice-based cryptoschemes, NTRU has had far less previous work done when it comes to hardware implementations. The main reason for this is the fact that NTRUEncrypt and NTRU-HRSS-KEM were merged somewhat recently, which reduced the relevance of the work that has been performed on these older schemes. The current version of NTRU has only existed for a few years and all previous implementations have to get major adjustments to work for this newest version. Nevertheless, work that has been performed on NTRUEncrypt [70] or NTRU-HRSS-KEM can still be used, since many of the functions that are required, such as polynomial multiplication, have not changed much from these older versions.

When it comes to the round 3 NTRU submission, Qin et al. (2021)[71] have designed a full hardware implementation of both the encryption and decryption for an (n,p,q) parameter set of (821, 3, 4096). Additionally, while other implementations are software and hardware co-design, the work of Wera, M.[72] demonstrates polynomial multiplication in NTRU using NTT when one of the polynomials is ternary.

This leaves room for a hardware implementation using the (n,p,q) parameter set of (701,3,8192), since this is the only recommended NTRU-HRSS parameter set and therefore requires a different lift function than the other parameter sets ($\mathbf{m}$ mapped to $\mathbf{\Phi_1} \cdot \underline{\text{S3}}\ (\mathbf{m}/\mathbf{\Phi_1})$). It should be mentioned again that this parameter set has the largest value of q of the parameter sets and therefore also the most computation heavy polynomial multiplication.

Multiplications in the NTRU algorithm

When it comes to polynomial multiplication there are several different categories, some of which specifically for polynomial multiplication on quotient rings. The most intuitive way to multiply two polynomials is to use the Schoolbook algorithm[73] with time complexity of $\mathcal{O}(N^2)$, however, this not only has a large time complexity, but also requires a large amount of multiplications. Several divide and conquer algorithms exist to reduce the time complexity and amount of multiplications required:

- k-way Toom-Cook with time complexity of $\mathcal{O}(N^{log(2k-1)/log(k)})$[74].
- Karatsuba with time complexity of $\mathcal{O}(N^{1.58})$[75].
- Schönhage-Strassen with time complexity of $\mathcal{O}(n \cdot log(n) \cdot log(log(n)))$[76].

However, these algorithms are still costly when implemented in hardware when working with the large polynomials in lattice-based cryptoschemes. Braun et al. (2018)[70] have demonstrated a convolution-based polynomial multiplication for NTRUEncrypt that takes advantage of the quotient ring structure and the

ternary polynomial input, reducing the time complexity to $\mathcal{O}(N)$. Despite the fact that this only works for a multiplication with a ternary polynomial, which is three of the four multiplications required in the round 3 NTRU submission, it is possible to reuse this hardware for the last multiplication as well.

There also remains the option to use NTT, or regular FFT transforms, however the parameter set of NTRU is not ideal for this. For regular FFT, a high level of floating point precision is required to retain a valid encryption and decryption output, increasing the area cost of an FFT block by a large amount. The requirements for NTT are to have polynomials of length n, where n is a power of 2, and a modulo q for which $q = 1 \bmod 2n$ holds. Neither of these conditions are true for the NTRU parameter sets, which means it requires additional transformations to use NTT. Even though Wera, M.[72] has shown that NTT can be performed on NTRU, specifically for the cases with ternary polynomials, more additional hardware would be required to perform NTT on all four required polynomial multiplications due to the fact that one of them does not have a ternary polynomial input.

4.2. System overview

The hardware implementation design can be divided into several different states that, while being separated from each other in functionality, are all reusing the hardware blocks that are present in the design. Figure 4.1 shows the high-level finite-state machine that splits the design into these different sections, where the right functionality can be selected by sending a matching mode signal to the hardware.

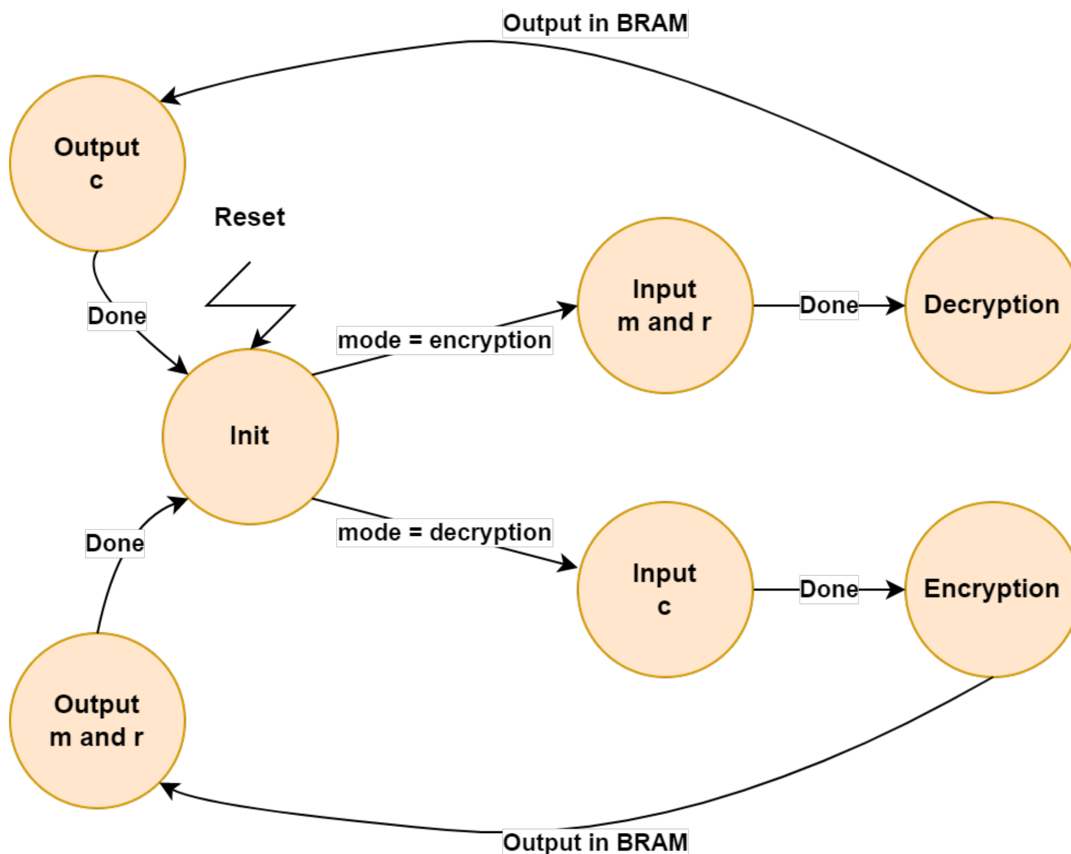


Figure 4.1: High-Level state machine overview of NTRU hardware functionality

Before going into each of these states and the hardware blocks that they are using in more detail, a brief overview of each function can be seen below:

- **Input:** The input states allow an outside source to interact with the internal memory. By giving a BRAM data input, new values of **r**, **m**, and **c** can be loaded into the BRAM. In the encryption input state a new value **r** and **m** is given to the hardware and in the decryption input state a new value of **c** is given to the hardware.

- **Encryption:** The encryption state will go through the steps of the encryption algorithm, first performing a polynomial multiplication between input **r** and public key **h**, followed by lifting input **m** and adding this to the multiplication result to receive output **c**.
- **Decryption:** The decryption state goes through the steps of the decryption algorithm, this is done by performing two polynomial multiplications in sequence, both followed by an additional reduction. The output of this multiplication, **m**, is then lifted and subtracted from the ciphertext input **c**. This result is then multiplied with the secret key $\mathbf{h_q}$ and reduced once again to receive output **r**.
- **Output:** The output states will output the ciphertext **c** after encryption, or output **r** and **m** after decryption. If the fail flag has been set during the checks performed in the decryption the output **r** and **m** will be replaced by zeroes.

Even though the input state is mostly separate and only interacts with the BRAM, it can be seen that the encryption and decryption states have very similar functionalities and will therefore reuse the same hardware blocks. It should be noted that the multiplications in the decryption algorithm have additional requirements and therefore additional logic will need to be added, which will be discussed in section 4.5.

4.3. Input State

The Input state allows for the entering of a new **r**, **m**, and **c** input and storing it into the BRAM so that it can be used for the upcoming encryption or decryption process. As shown in figure 4.1, there is always an input state before both the encryption and decryption, with the main difference being which input values are being stored. Because **r**, **m**, and **c** are all an equal amount of coefficients this means that the input state for the encryption will take longer than the input state for the decryption because they are all using the same BRAM. However, since **r** and **m** are both ternary polynomials, these values can be stored more efficiently in the BRAM.

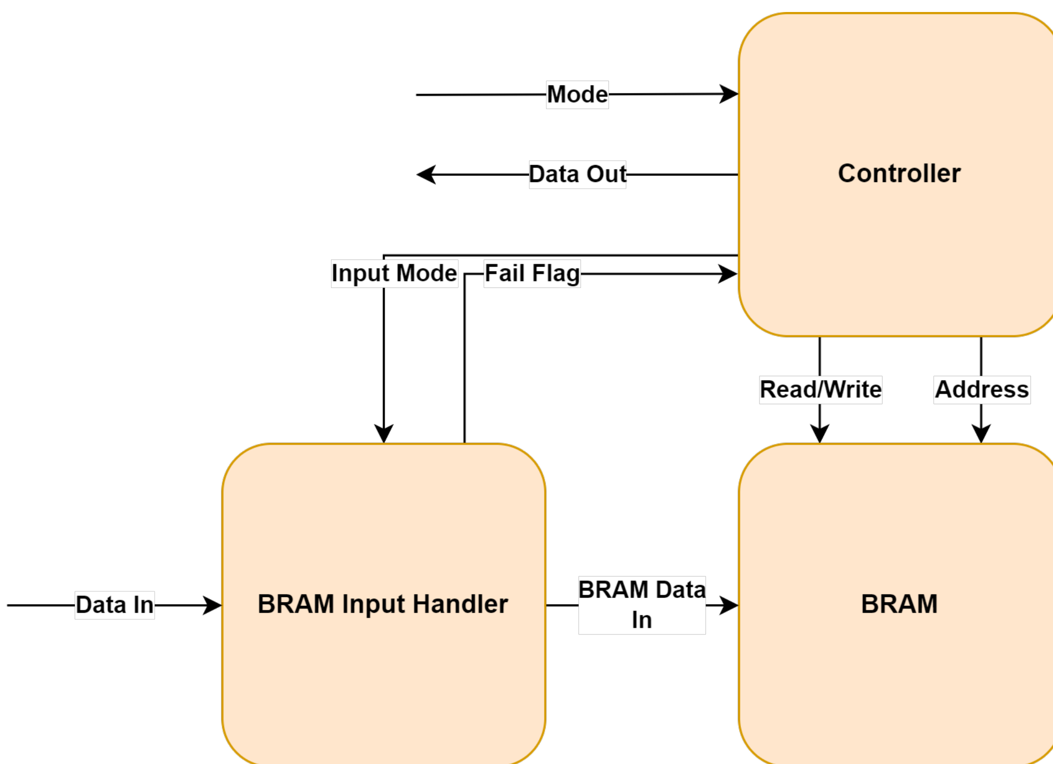


Figure 4.2: High-Level overview of the blocks used by the Input state

Figure 4.2 shows the three blocks that are used in the Input state and the dataflow between these blocks. For just implementing the Input State the BRAM Input Handler block is not required, however due to the fact that multiple blocks have to store data inside the BRAM in the overall system, some form of input handling is required.

4.3.1. Control signals for Input State

The controller sends the correct address and read/write mode to the BRAM while any form of interfacing with the BRAM is happening. In the Input state specifically, the controller sets the Input Mode of the BRAM Input Handler to take the external Data In value and read this into the BRAM. To indicate when to input the coefficients of **r**, **m**, and **c**, the controller will output a GiveInput signal. Lastly, corresponding to the NTRU algorithm, the fail flag will also be output by the controller.

4.3.2. BRAM Structure

The BRAM is set up to store all polynomials on a set address, these polynomials not only include the input and output signals, but the polynomials generated by the key generation part of the algorithm as well. There are two sizes of polynomials that need to be stored in the BRAM, the ternary polynomials (mod **3**) and the larger polynomials (mod **q**), and to use the same BRAM for all these polynomials the width of each address is set to **q**. Only one address can be read per clock cycle and most polynomials stored in the BRAM are read sequentially when used in the Encryption and Decryption states, therefore by storing one value per address this simplifies the interfacing between the different blocks. An exception to this is the value of **m**, since the Lift input block requires three coefficients as an input, and because **m** is a ternary polynomial it is possible to store three coefficients on each address without introducing any other issues.

Figure 4.3 shows the allocation of the BRAM for each address.

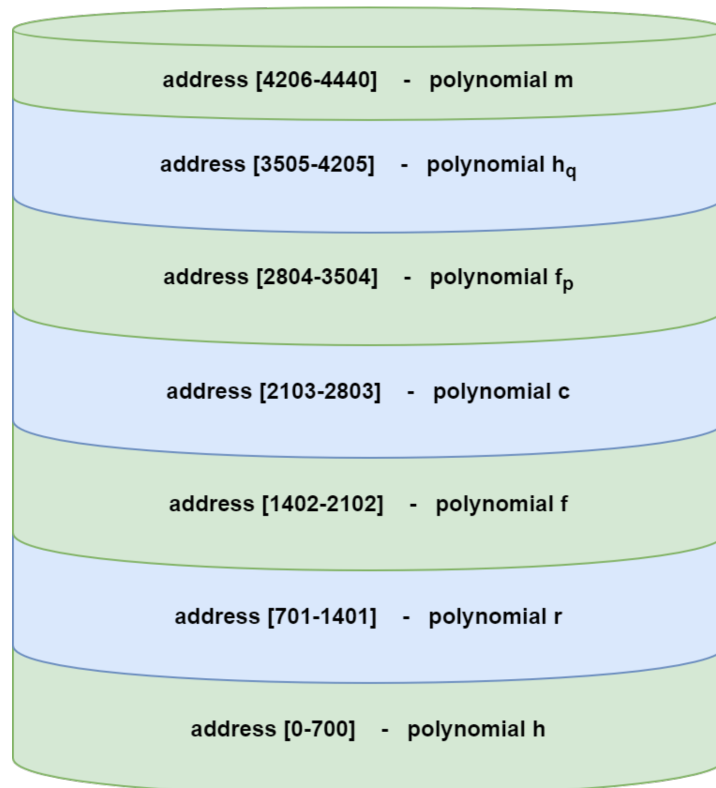


Figure 4.3: Overview of the BRAM address allocation for both reading and writing polynomials.

Shown by Figure 4.3, each polynomial takes up 701 addresses which is equal to the polynomial length n, with the exception of polynomial **m** which takes up $n/3 = 234$ address spaces instead.

4.3.3. BRAM Input Handler

The BRAM Input Handler performs two tasks:

- Based on the Input Mode signal it will switch between the different inputs and potentially add two inputs together to compute the correct output.
- Perform a validity check on the decryption input and output and set a corresponding fail flag.

The different outputs based on the Input Mode can be seen in table 4.1, as can be seen from the table the output data is always equal to either **r**, **m**, or **c**.

Input Mode	Output Calculation	Output in the Algorithm
00	Data Out = External Data In	**r**, **m**, or **c** (User Input)
01	Data Out = "0000000" & Ternary Register Outputs	**m**
10	Data Out = Lift Output + Convolution Output	**m'** + (**r** · **h**) = c
11	Data Out = Reduced Output	**r**

Table 4.1: Multiplexer table showing the relation between the Input Mode signal and the corresponding output signals

Because there are two validity checks in the NTRU algorithm, one check is performed on **c** during the Input state before the decryption, and another is performed at the end of the decryption when **r** is stored in the BRAM. The check on **c** ensures that **c** = 0 mod (q, ϕ_1) by adding all the coefficients that come in sequentially and checking if the final sum is equal to 0. For **r** the check ensures that each coefficient of **r** is 0, 1, or 2, confirming that the output **r** is a ternary polynomial. Additionally it also checks if the last coefficient is 0.

4.4. Encryption

The encryption state performs the full encryption step of the NTRU algorithm, turning polynomials **r** and **m** into ciphertext **c**. The high-level block diagram can be seen in Figure 4.4.

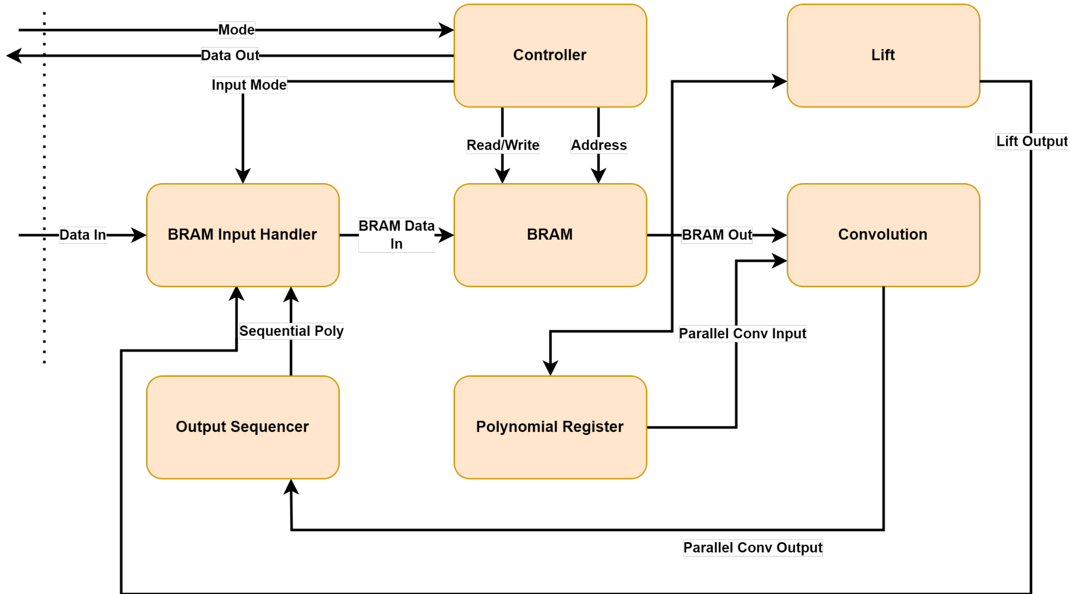


Figure 4.4: High-Level overview of the blocks used by the Encryption state.

In addition to the blocks that were introduced in the input state, which are needed to input **r**, **m**, and the public key **h** that is used in the encryption, additional blocks are added to perform the encryption:

- Convolution: This block performs the polynomial multiplication that is required in the encryption.
- Lift: this block performs the lift function $m' = Lift(m)$.
- Polynomial Register: To perform multiplication in parallel, all coefficients of one polynomial have to be available somewhere and the polynomial register is used for this.
- Output Sequencer: Converts the parallel multiplication output into a sequential input s that it can be stored in the BRAM.

4.4.1. Convolution Module for Encryption

The Convolution module is the main block for computing the initial multiplication between polynomials. This form of multiplying was demonstrated by Braun et al. [70] in an NTRUEncrypt hardware implementation and while some additions must be made for the Decryption process, for the Encryption state, the demonstrated implementation can compute **r** $\cdot$ **h**. Figure 4.5 shows the base version of the convolution block.

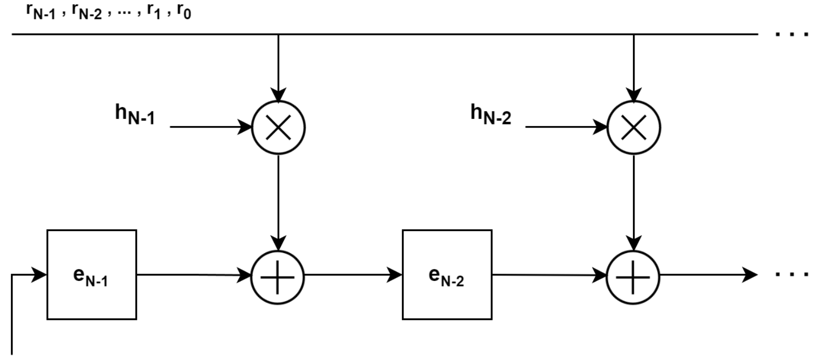


Figure 4.5: Schematic of the first two registers in the convolution shift register.

The implementation by Braun et al. [70] uses a shift register of size N to multiply two polynomials of size N with complexity $\mathcal{O}(N)$, as opposed to $\mathcal{O}(N^2)$. By using N shift registers **e**, giving input polynomial **h** in parallel, and input poly polynomial **r** in series, N multiplications can be performed at once every clock cycle. As polynomial size increases an N amount of multiplications can become costly, however, in the NTRU algorithm most polynomial multiplications involve at least one ternary polynomial. The Convolution block takes advantage of this by replacing the required multiplication by an addition, this can be done because the values of input **r** can only be {0,1,2} and these inputs would result in the following addition:

- **r = 0:** $e_{N-2} = e_{N-1}$
- **r = 1:** $e_{N-2} = e_{N-1} + h_{N-1}$
- **r = 2:** $e_{N-2} = e_{N-1} - h_{N-1}$ which, in mod q is equal to, $e_{N-2} = e_{N-1} + !h_{N-1} + 1$

Since all the outputs of the multiplication are in mod q and mod q is chosen to be a power of 2 in the NTRU algorithm, the modulo q operation is performed for free by using a register of size q. Because of these properties, the logic between the registers can be replaced by an adder and multiplexers that are driven by the MSB of the ternary polynomial input. This implementation of the new MAU block can be seen in Figure 4.6.

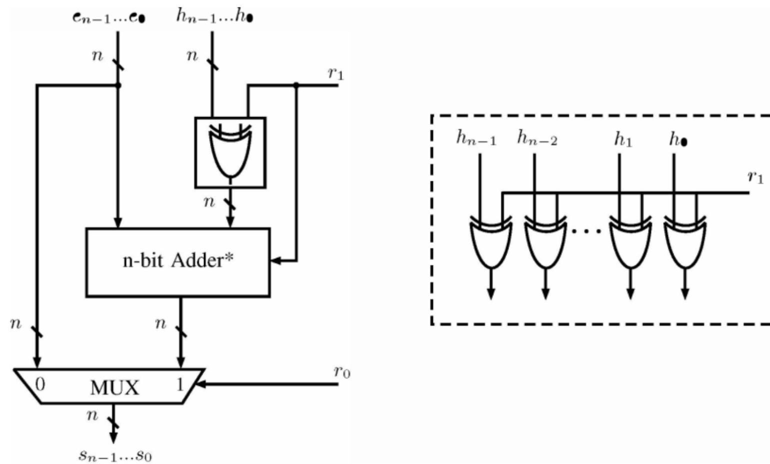


Figure 4.6: Schematic of the MAU implementation in the convolution block.[77]

The MAU block shown in Figure 4.6 is an efficient way to perform the n additions and multiplications required each clock cycle that has been demonstrated by Braun et al. [70] and was originally proposed by Bingxin Liu and Huapeng Wu [77].

4.4.2. Polynomial Register

Although the serial input for the Convolution block can be read directly from the BRAM, since it outputs one coefficient per clock cycle, the parallel input has to be fully available. Because this parallel input is in mod q, a set of N registers with width q are required. Therefore, as an extension of the Convolution block, the Polynomial Register block is used, which is directly connected to the BRAM output.

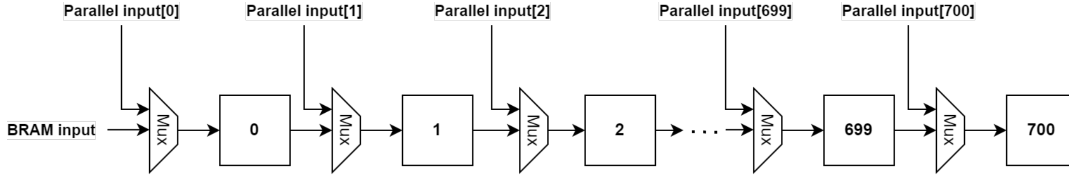


Figure 4.7: Schematic of the polynomial register implementation.

The Polynomial Register can be seen in Figure 4.7 and it is a standard shift register, which means that the BRAM can output one coefficient per cycle and slowly fill the registers until all the coefficients are available for use by the Convolution block. Additionally, all the registers have a second possible input based on a control signal from the controller which is used in the decryption state. Line 2 and 3 of the decryption process described in Figure 3.2 are two subsequent multiplications, where the first multiplication result is used as the parallel polynomial input of the second multiplication. To eliminate the need for additional registers this additional Polynomial Register input allows that intermediate input to be stored in the same registers.

Because it takes N cycles to fill up the Polynomial Register due to the structure of the BRAM, the time it takes to compute a multiplication would go up to $\mathcal{O}(2N)$, however, part of this overhead can be removed by starting to fill the Polynomial Register while other operations that do not use the BRAM are running, as it is separate from all other blocks.

4.4.3. Lift Operation

The Lift Operation takes input **m** and returns output $\phi_1 \cdot \mathbf{S3}(\mathbf{m}/\phi_1)$ by following Algorithm 2. As can be seen from Algorithm 2 line 3, there is an expensive dot product operation that involves the multiplication of two N-coefficient long vectors. However, Hülsing et al.[66] have shown that, by using the near periodicity of the polynomial z, the complexity of this computation can be severely reduced in this algorithm. Additionally, this periodicity lends itself towards a low-area hardware implementation of this dot product calculation.

Algorithm 2 Lift Function Algorithm

Input: v
1: $z = [1/\phi_1]_p$
2: **for** $i = 0$ **to** $i = p - 1$ **do**
3: $\quad a_i = \langle x^i \bar{z}, v \rangle$
4: **end for**
5: **for** $i = p$ **to** $i = n - 1$ **do**
6: $\quad a_i = a_{i-p} - \sum_{j=0}^{p-1} v_{i-j}$
7: **end for**
8: $a_0 = a_0 - a_{n-1} \bmod p$
9: $b_0 = -a_0$
10: **for** $i = 1$ **to** $i = n - 1$ **do**
11: $\quad a_i = a_i - a_{n-1} \bmod p$
12: $\quad b_i = a_{i-1} - a_i$
13: **end for**
Output: b

Figure 4.8 shows an overview of the Lift block, it can be seen that there are two register blocks and several

computation blocks for the Lift function. Before discussing the implementation and choices behind these blocks, it is important to go over the implementation of the dot product, because this implementation largely drove the design of all the other blocks within the lift function.

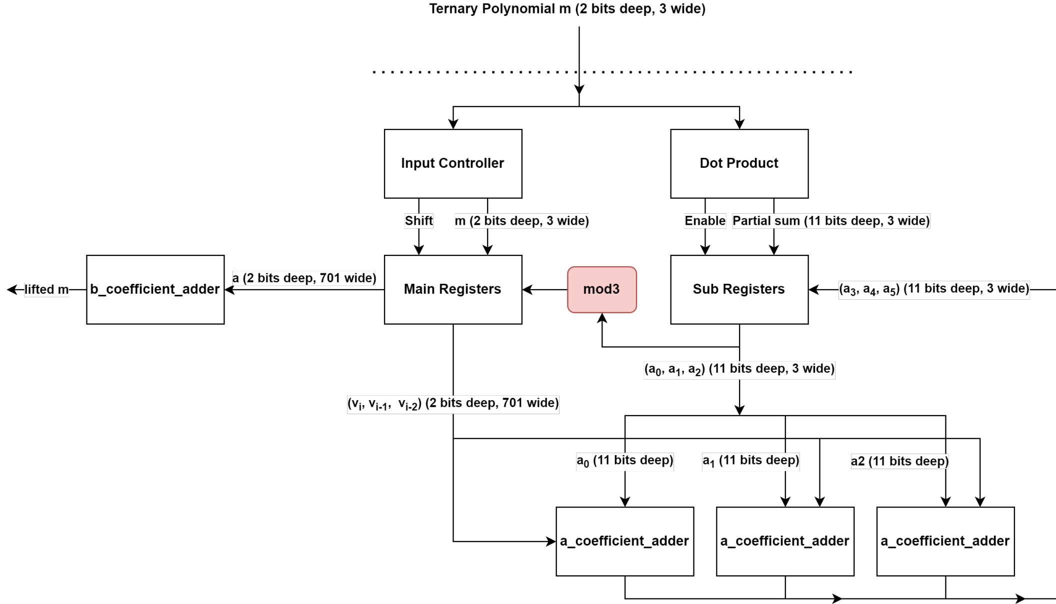


Figure 4.8: Overview of the Lift block components and their dataflow.

Dot Product in Lift

The dot product is an expensive operation with a large amount of multiplications and additions which is used to compute the values of a_0, a_1, and a_2. These are then used to compute the remaining values of **a** (line 6, Algorithm 2), however, like mentioned above, it is possible to take advantage of the periodicity of one of the vectors. Based on the efficient computation of Lift by Hülsing et al.[66] vector $\bar{z}$ can be expressed as:

$$\bar{z} = -(t+1) + \sum_{i=1}^{n-1} t(i-1)x^i \text{ (mod p)} \tag{4.1}$$

Where:

$$t = 3 - (\text{n mod } 3) \tag{4.2}$$

Because the value of **t** in equation 4.1 is only related to the polynomial length, which is a static value in each implementation, it can be seen that a coefficient of $\bar{z}$ is only dependant on the vector index and this static value **t**. Furthermore, the result is in mod p, which is the modulo that you are lifting from, p = 3. This means that, with the exception of the first two coefficients, given a coefficient of $\bar{z}$, the next coefficient will always be:

$$\bar{z}(i) = \bar{z}(i-1) + t \text{ (mod 3)} \tag{4.3}$$

Algorithm 2 also multiplies this vector $\bar{z}$ by x and x^2 to compute the values of a_1 and a_2 respectively. For the most part this can be interpreted as a right shift on the periodic vector $\bar{z}$, with one or two additional computations required to compensate for the two values that have been shifted out from the last two coefficients. Table 4.2 shows the first few coefficients of the three used polynomials and from this it can be seen that starting from i=3, the three of the vectors repeat their coefficients with a period of 3.

Vector	i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9
$\bar{z}$	1	0	1	2	0	1	2	0	1	2
$x \cdot \bar{z}$	0	1	0	1	2	0	1	2	0	1
$x^2 \cdot \bar{z}$	0	0	1	0	1	2	0	1	2	0

Table 4.2: Periodicity of the three vectors $\bar{z}$, $x \cdot \bar{z}$, and $x^2 \cdot \bar{z}$

To take advantage of this periodicity the following steps have been implemented:

- The lift function will take three coefficient inputs of **m** at once so that the same logic can be reused for almost every input.
- The exception to this, the first two inputs, are not periodic but always the same multiplications and can therefore be replaced by a LUT.
- The Sub Registers block is introduced to store the intermediate values of a_0, a_1, a_2, and the final value of these coefficients.

In addition to these points, the polynomial is of size n = 701, which, since it is a prime number, can not be divided by 3. To still give three coefficient inputs of **m** at once, the polynomial **m** has an additional coefficient added that is always zero. This means that the first input will be $\{m_1, m_0, 0\}$ and the second input is then $\{m_4, m_3, m_2\}$. As a result of this change, the second input is not periodic and has to be replaced by a LUT similarly to the logic for the first input.

Figure 4.9 shows a diagram of the sub registers that are used to compute a_0, a_1, and a_2. These registers are 11 bits wide and will add their stored value to the new value whenever an enable signal is given, providing the final values of a_0, a_1, and a_2 once all the input values of **m** have been given.

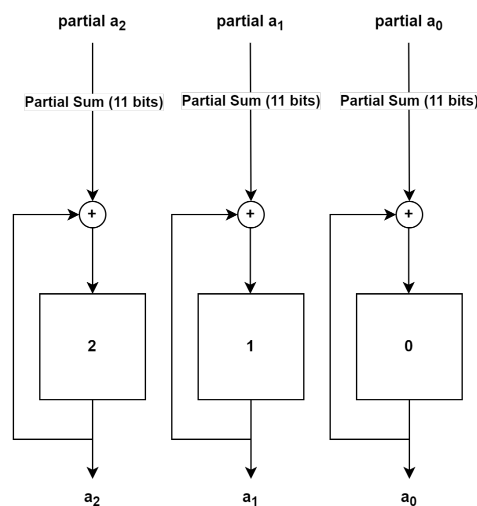


Figure 4.9: Sub Lift Registers design for computing dot product.

Computing vector a

From Algorithm 2 it can be seen that the dot product is only used to compute coefficients a_0, a_1, and a_2, while the remaining coefficients of **a** are calculated by using these first three coefficients of **a** and the original input **v**. Because the input of the Lift function comes from the BRAM, it would slow down the encryption process if this input has to be read from the BRAM for this step as well, since the BRAM could be reading other values at this time. To prevent this, the input **v** is stored in a shift register within the Lift block, specifically the Main Register in Figure 4.8. While the dot product computation is happening, the inputs are inserted into the Input Handler as well, which then inputs accordingly to the shift register along with an enable signal to start shifting.

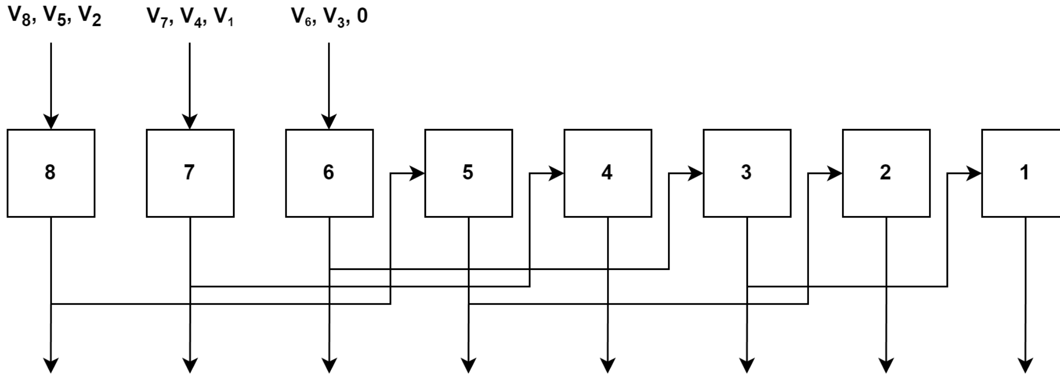


Figure 4.10: Main Lift Registers design to store vector **v** and **a**.

A schematic overview of the Main Register block can be seen in Figure 4.10 showing that it is shifting three places at a time, since there are three coefficient inputs each clock cycle. To accommodate for the extra input zero that is used by the dot product the Main Register is N+1 long and the last register is ignored for future computations.

Adding these registers to store an entire ternary polynomial comes at non-negligible area cost, however, these registers can be reused to store the coefficients of **a** as well. Line 8 and 11 of Algorithm 2 indicate that the final coefficient of **a** is required to perform a final computation on each other coefficient, which demands the storing of all coefficients.

To compute the other values of **a**, three separate adder blocks are used along with the Main Registers and Sub Registers. Looking at Line 6 of Algorithm 2:

$$a_i = a_{i-p} - \sum_{j=0}^{p-1} v_{i-j} \tag{4.4}$$

It can be seen that to compute a_3 the coefficients a_0, v_3, v_2, and v_1 are required, while to compute a_4 the required coefficients are a_1, v_4, v_3, and v_2. This shifting property repeats for all coefficients and, since input **v** is stored in a shift register already, a lot of logic can be reused to calculate all coefficients of **a**. Since three coefficients of **a** are available in the Sub Registers, and the Main Registers shift by three places at a time, the choice was made to compute three coefficients of **a** per clock cycle by reusing all the present registers.

The three a coefficient adders in Figure 4.8 are used to compute $\{a_3, a_4, a_5\}$, $\{a_6, a_7, a_8\}$, $\{a_9, a_{10}, a_{11}\}$ etc. until all coefficients have been calculated. This is done by getting the five required inputs of **v** from the Main Register and the three required inputs of **a** from the Sub Registers. The coefficients of **a** that are calculated this way are then stored in the Sub Registers so that they can be used to get the next three coefficients of **a**. Additionally, since the input polynomial **v** is slowly being shifted out of the Main Register during this operation, the used coefficients of **a** are input to the Main Register so that they can be used for the remaining part of the algorithm. Since these coefficients are 11 bits large, a modulo 3 operation is performed so that they fit inside the Main Register.

At the end of this step the Main Register will have shifted out all of input **v** and have been replaced by the fully computed vector **a**.

Computing the output vector

To get the final output of the Lift function, the last few steps of the algorithm still have to be implemented. Line 8-12 of Algorithm 2 involve subtracting the last coefficient of **a** from each coefficient, and computing output vector **b**.

Since the entire vector **a** is present in the Main Register it can be calculated in parallel, resulting in a parallel output of the Lift function. By performing this last step in parallel, it only takes one clock cycle at the cost of requiring more area.

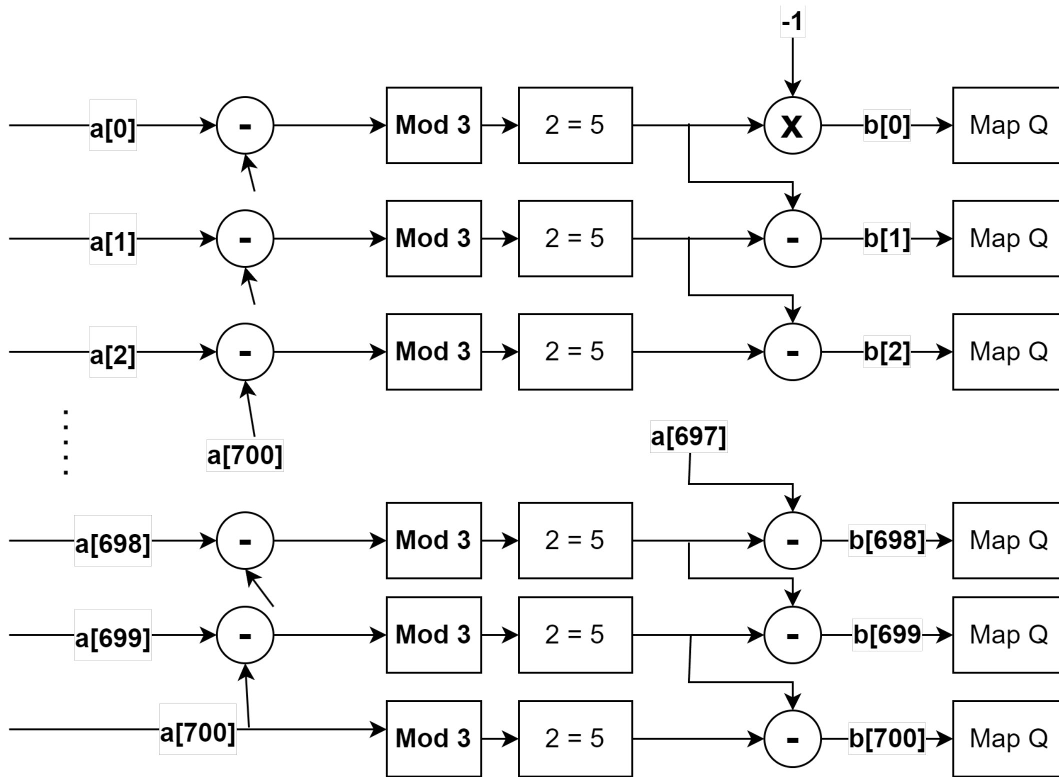


Figure 4.11: b coefficient adder block design.

Figure 4.11 shows the final block of the Lift function. Step-by-Step each coefficient first has a_{N-1} subtracted from it, followed by a modulo 3 operation and a mapping. This mapping is performed so that when the subtraction afterwards is performed, there will be no combination of two inputs that give similar results. In software a mapping to Q-1 would be performed here, however, since Q is 2^{13} this would require much larger adders, therefore a smaller mapping is performed instead. To compensate for this, the output values are mapped a second time after the subtraction so that the desired Q-1 mapped output is achieved.

The final output of this block will be the desired polynomial mod 3 lifted to mod Q, which is used to compute the Encryption output **c**. Even though implementing this final block to be a computation in parallel is simpler and faster than doing it in series, the area cost of the final block is substantial and this will be evaluated in Chapter 6.

4.5. Decryption

The decryption state performs the full decryption step of the NTRU algorithm, turning ciphertext **c** into the polynomials **r** and **m**. The high-level block diagram can be seen in Figure 4.12.

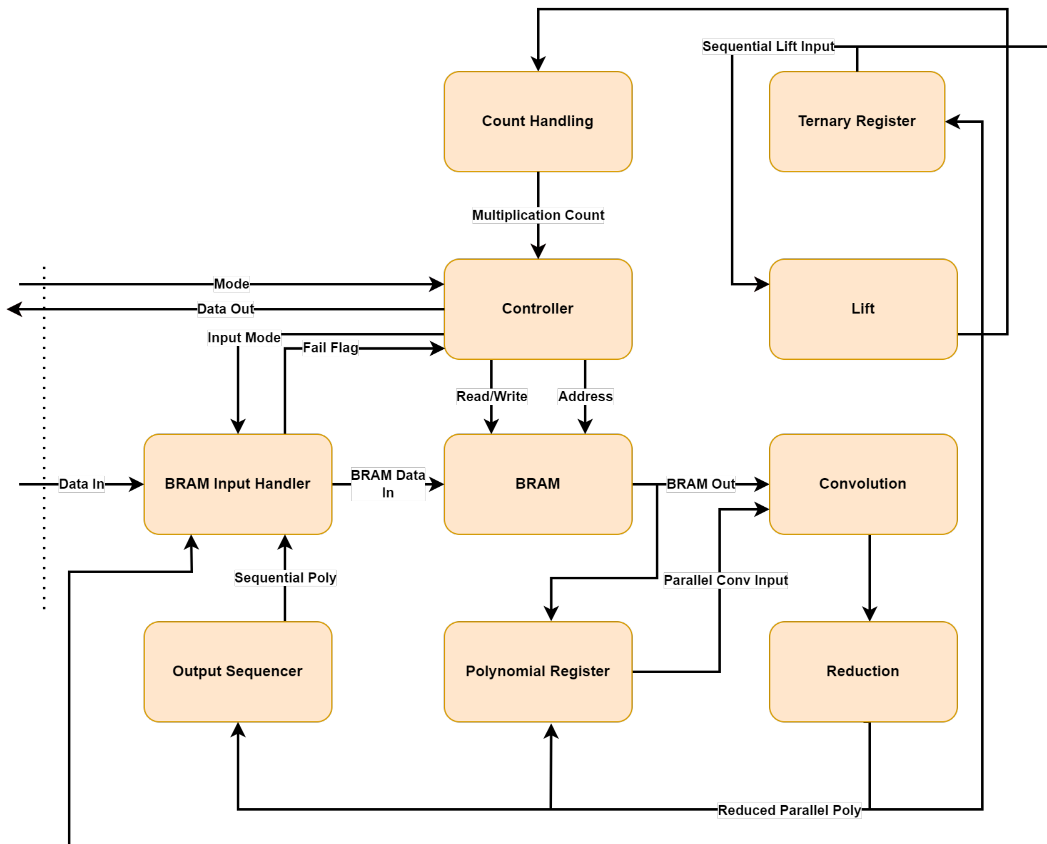


Figure 4.12: High-Level overview of the blocks used by the Decryption state.

The Decryption State reuses a lot of the blocks that are used in the Encryption State, since the NTRU algorithm uses the Lift and Polynomial Multiplication step in both Encryption and Decryption. The main difference in approach is that the Decryption State has to perform three multiplications, opposed to the single multiplication in Encryption, and a Polynomial Reduction has to be performed between these multiplications. Additionally, the two validity checks that have been discussed in Section 4.3 are performed as well.

The new blocks that are used in this step are:

- Reduction: This block performs modular reduction which is required between multiple polynomial multiplications that are done of different quotient rings.
- Ternary Register: Stores the intermediate output **m** without accessing the BRAM.
- Count Handling: Additional block to support the final multiplication of the decryption which has no ternary polynomial input.

4.5.1. Ternary Register

Figure 3.2 shows that in Line 3 a Polynomial Multiplication is performed to retrieve **m** which is then immediately lifted to **m'** in the next step. Polynomial **m** is retrieved in parallel and the Lift function requires a serial input, therefore some additional interfacing would be required. Due to the fact that **m** is a Ternary Polynomial the cost of storing it in a register is rather small and using a shift register is a simple way to turn this into a serial input for the Lift function. A Ternary Register is introduced to do this, which is a shift register that is 2 bits wide and 702 coefficients deep to accommodate all of **m**. Similarly to the shift registers used in the Lift function, this register shifts three places at a time so that it can provide the three inputs that the Lift function requires each clock cycle

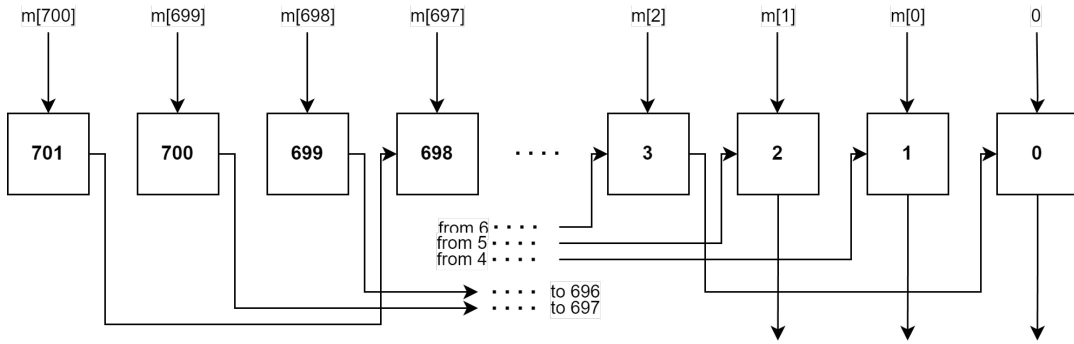


Figure 4.13: Schematic of ternary register implementation.

Figure 4.13 shows the schematic of the ternary register implementation. It can be seen that the first register is always loaded with a value of zero to accommodate for the padding required for the Lift function, resulting in a shift register that is $n+1$ long.

An added advantage of converting **m** from a parallel input to a serial output like this is that **m** can be stored into the BRAM at a later time, since it will remain in the Ternary Register for the remainder of the Decryption operation. Which allows the BRAM to start loading polynomials that are required for the subsequent steps while the Lift function is running.

4.5.2. Convolution Module for Decryption

Even though the base Polynomial Multiplication has already been discussed in Section 4.4, additions must be made to this block to support the multiplications that are required for the Decryption. As can be seen in the NTRU algorithm (Figure 3.2), there are three multiplications, and all three of them have minor differences that must be taken into account.

$$\mathbf{a} \leftarrow (\mathbf{c} \cdot \mathbf{f}) \bmod (q, \phi_1 \phi_n) \tag{4.5}$$

The first multiplication (Eq 4.5) is using the same modulo and quotient ring that is used in the Encryption and can therefore be performed by using the same logic that was used in the Encryption State. However, the multiplication right after that is using a different modulo and quotient ring:

$$\mathbf{m} \leftarrow (\mathbf{a} \cdot \mathbf{f}_p) \bmod (3, \phi_n \tag{4.6}$$

Therefore, the output **a** of the first multiplication must first be reduced to be on the same quotient ring before it can be used in the second multiplication as an input. The reduction is handled by the reduction block and will be discussed in section 4.5.3. Additionally this second multiplication is performed modulo 3 which needs to be considered when performing the convolution. Modulo q is a free operation because it is a power of two and by limiting the register size it will automatically reduce every coefficient by modulo q, but this is not the case for the modulo 3 operation. The modulo 3 operation will have to be performed between registers and to accommodate for this additional control logic is added to the Convolution block. This additional logic allows choosing between either using the MAU block and an additional modulo 3 block, or just using the MAU block exclusively, depending on the multiplication that needs to be performed.

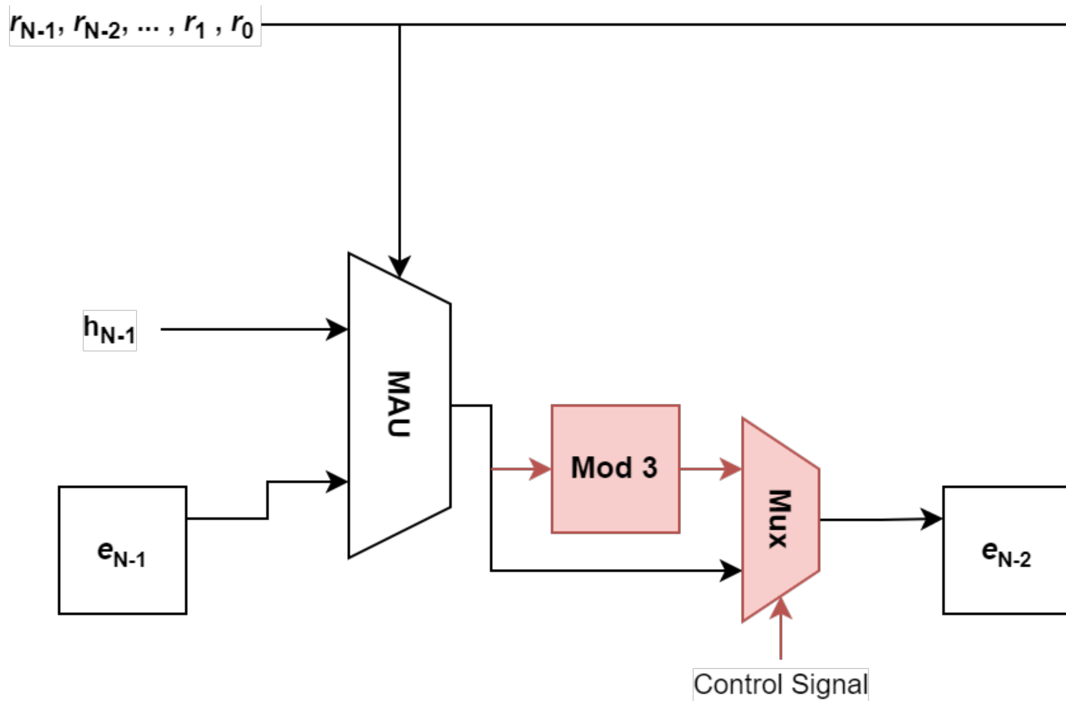


Figure 4.14: Convolution Block additions for second multiplication, additions are shown in red.

Figure 4.14 indicates the changes that need to be made compared to the original convolution setup shown in Figure 4.5, where the adders have been replaced by a singular MAU block. As can be seen from the figure a control signal from the Controller will select which multiplication needs to be performed by multiplexing the correct output from the logic between the registers. Since the output of the MAU block is used by both multiplications, this selection requires no additional logic besides a multiplexer between each register.

13-bit multiplication Convolution

The final multiplication, shown in Equation 4.7, requires additional discussion

$$\mathbf{r} \leftarrow ((\mathbf{c} - \mathbf{m'}) \cdot \mathbf{h}_q) \bmod (q, \phi_n) \tag{4.7}$$

Unlike the other polynomial multiplications, this multiplication does not have a ternary polynomial as an input. Due to this, the relatively cheap MAU operation can not be used and an actual multiplication needs to be performed. There are several ways to approach this, which has been discussed in Section 4.1, however, the Convolution block is already there for the other multiplications and reusing it would mean that much less additional area is needed compared to other methods.

There are several ways to approach this which ultimately ends up being a trade-off between area and execution time. A 13-bit by 13-bit multiplication needs to be performed between each register and the fastest way to do this would be to use DSP modules between each register, however, the area cost of using N=701 DSP modules is quite large. This area impact can be lessened by reducing the amount of DSP modules and performing each Convolution cycle in multiple steps, however this requires some additional logic.

Even though using several DSP modules and adding additional logic to balance the area and execution time trade-off can give a desired result, there is also a method to perform this multiplication without DSP modules. By adding additional logic between the registers so that the input and output of the same registers are connected, the MAU operation can be used multiple times in sequence to act like a multiplication.

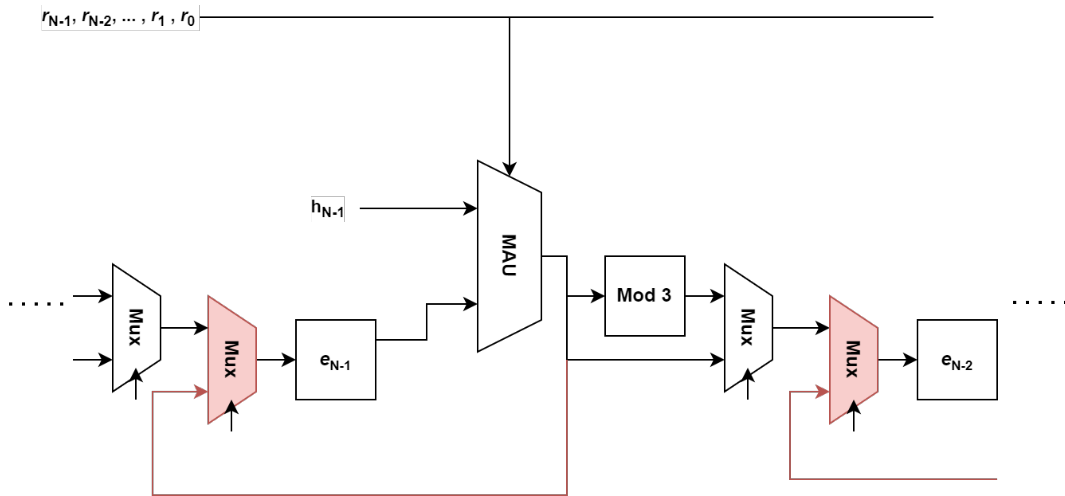


Figure 4.15: Convolution Block additions for third multiplication, additions are shown in red.

Figure 4.15 shows the proposed design. By adding these multiplexers and providing the correct control signals the MAU operation can be performed multiple times before the output is shifted to the next register. Using this implementation the parallel input will remain unchanged while an additional block will keep track of the sequential input. The sequential input will get a constant value of 1 while the "multiplication" is being performed and the additional block, the Count Handling, will count how many additions need to be performed to match the required multiplication. This turns an $\mathbf{r} \cdot \mathbf{h}$ multiplication into $\mathbf{r}$ times an $\mathbf{r} + \mathbf{h}$ addition.

4.5.3. Reduction

The reduction block performs the conversion between quotient rings that are required for subsequent multiplications to take place during the Decryption process. Similarly to the end of the Lift block discussed in Section 4.4.3, this operation is performed in parallel since it receives a parallel input from the Convolution block. The operation that is performed is subtracting the last coefficient from all other coefficients and optionally performing a modulo 3 operation. When reduction is performed between polynomial multiplications the modulo 3 block is always used, but the reduction that is performed after the third multiplication in the Decryption does not require a modulo 3 operation.

Figure 4.16 shows the Reduction block design

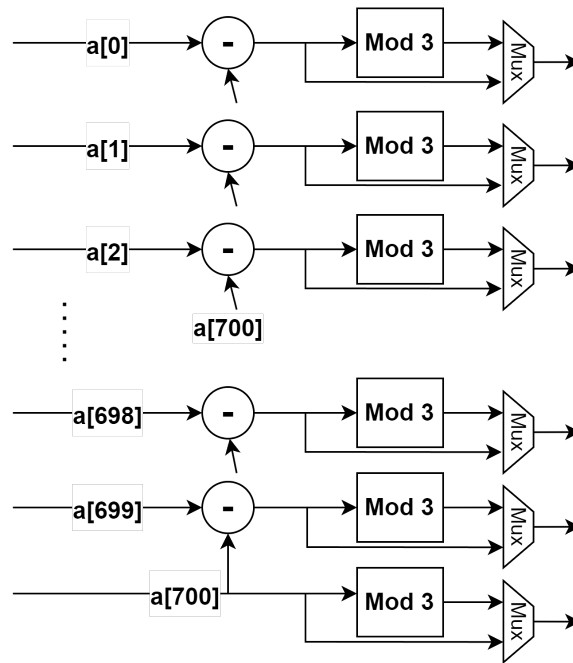


Figure 4.16: Reduction block design.

Although this block has been implemented to work in parallel due to the parallel input and output to reduce the amount of clock cycles, the area cost of this block is much higher than it would be if it were implemented to run in series and will be discussed in Chapter 6.

4.6. Output State

The Output state is the final step of both the Encryption and Decryption process and its purpose is to provide an output of **r**, **m**, or **c**. However, these outputs are all computed in different blocks and need to be routed accordingly at the end of Encryption and Decryption. Before **r**, **m**, or **c** can be given as an output they need to be stored in the BRAM and the BRAM Input Handler is set up to output **r**, **m**, or **c** depending on its input mode like shown in Table 4.1.

This requires the serializing of the parallel outputs of the Lift, Convolution, and Reduction blocks so that they can be inserted into the BRAM at the rate of one coefficient per clock cycle (three in the case of **m**).

In the case of the Convolution the registers can be shifted while giving a 0 as serial input, this will allow a single register to go through one coefficient per clock cycle providing a valid BRAM input. Both the Lift and Reduction contain parallel logic that is dependant on the last coefficient of a Polynomial, therefore shifting the registers in front would cause a different coefficient to be subtracted and this would generate an invalid output. By storing the last coefficient in an additional register that has a different enable signal, the output of both the Lift and Reduction can be shifted while retaining this last coefficient.

Once **r**, **m**, or **c** have all been stored in the BRAM depending on whether it is the Encryption or Decryption state, they can be given as an output. For the Encryption the output will always be **c** and for the Decryption the output is dependant on whether the Fail flag has been set throughout the decryption by performing the two checks described in Section 4.3.3. With the Fail flag set the output will be only zeroes, however, if the output is considered valid then it will be (**r**, **m**).

4.7. Full System Overview

Figure 4.17 gives an overview of the full implementation and how all the different blocks that have been described in this chapter are connected. To indicate which data is used by encryption or decryption different colours are used:

- Green connection: used by encryption only.
- Red connection: used by decryption only.

- Purple connection: used by both encryption and decryption.

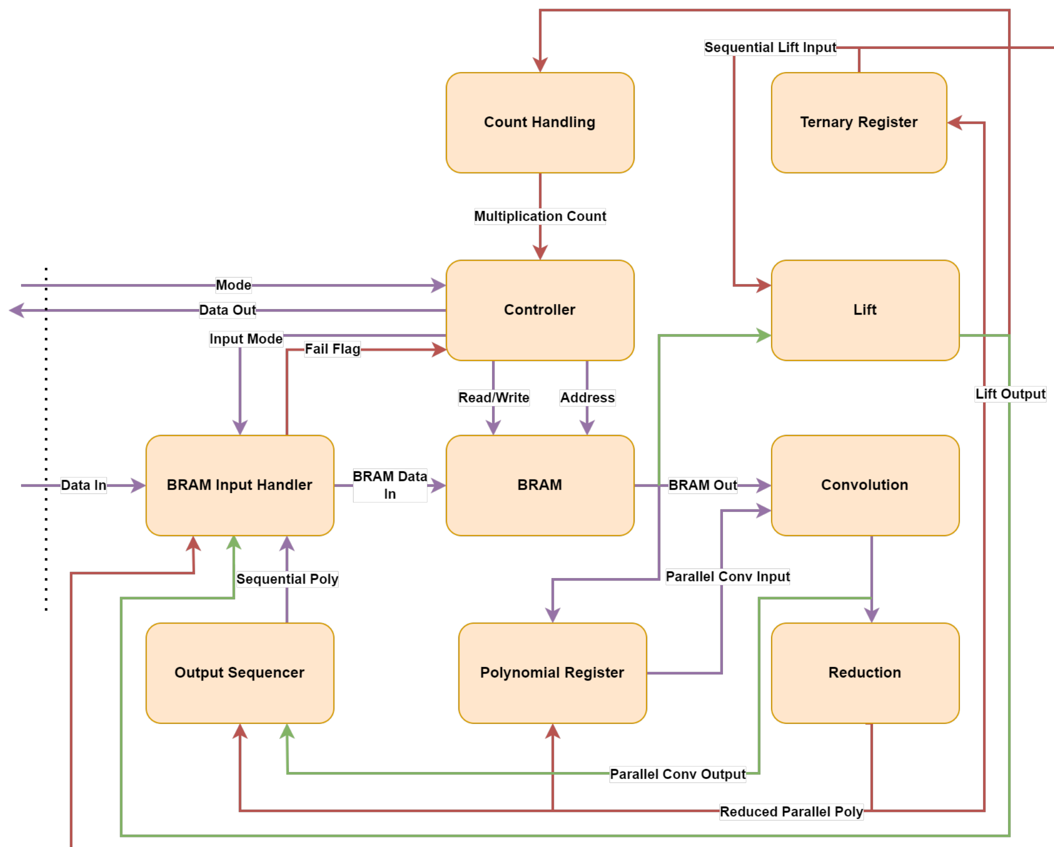


Figure 4.17: Full system overview.

It should be noted that there are no multiplexers shown in Figure 4.17 and therefore signals such as the **BRAM Out** signal are connected to three blocks at once. Since multiplexing an n by q $(701 \cdot 13)$ signal, or even just a q (13-bit) signal, can costs additional area, the decision was made to use enable signals for the functional blocks so that the multiplexing of signals is not required everywhere. This way registers and the BRAM can output to any blocks that uses their output and the controller block enables the functional blocks that need the signals at a specific point in time of the algorithm.

5
Results and Analysis

This chapter will cover the results and the discussion about these results. Section 5.1 will discuss the experimental setup that was used to perform testing, simulations, synthesis, and implementation of the hardware. Section 5.2 will discuss the area usage of the different blocks of the design and discuss amount of clock cycles that are required to perform the encryption and decryption. Section 5.3 looks at each step of the algorithm, highlighting the impact each step has on the execution speed and logic usage, as well as discuss the various simulations that were performed to validate the more complex blocks in the implementation. Section 5.4 will discuss these results, going over the hardware blocks where improvements will have the greatest impact on the amount of area used or amount of clock cycles required and compare the implementation with the reference software. Lastly, Section 5.5 will perform an analysis of the implementation security, highlighting blocks that contain private information and potential side-channel attack risks.

5.1. Experimental setup

All implementation, simulations, and measurements have been performed using Vivado ML 2022.1 [78] using the Kintex UltraScale+ KCU116 Evaluation Platform for all synthesis, implementation, and simulation results. For all simulations used to measure the amount of clock cycles the built-in Vivado Simulator was used. All implementation results are from the post-implementation summary and hierarchical view results that are provided after a succesful implementation. Similarly, all hardware synthesis results are from the post-synthesis summary that is provided after synthesis.

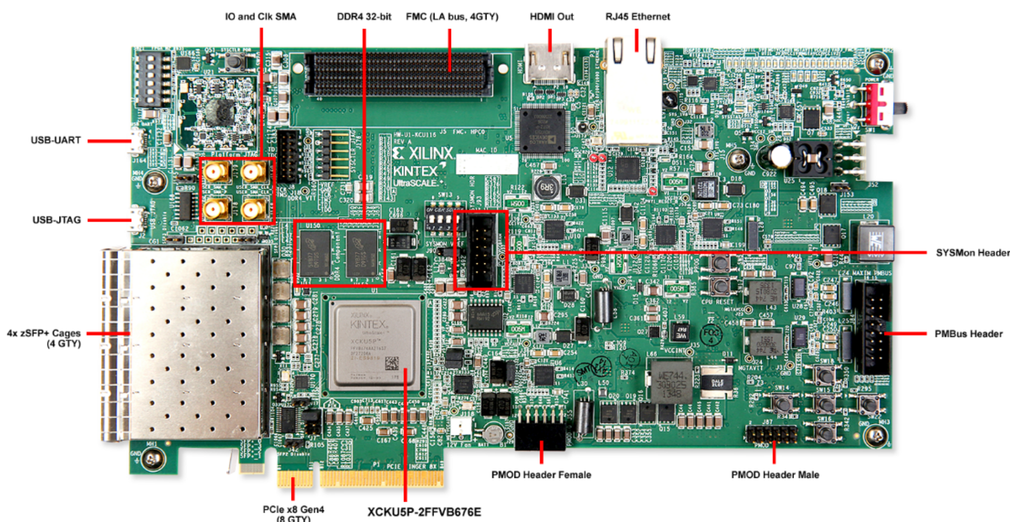


Figure 5.1: Xilinx Kintex UltraScale+ FPGA KCU116 Evaluation Kit Board layout. [79]

The Kintex UltraScale+ KCU116 Evaluation Platform has the following specifications:

- 280 IO ports.
- 216960 available LUTs.
- 433920 available FlipFlops.
- 480 available BRAM units.

5.2. Area Overhead Results

To give an overview of the area usage of the full implementation and the area usage of each block by itself, two sets of results will be used. The post-implementation results of the full block diagram with its hierarchical view is presented in Table 5.1.

Component Name	CLB LUTs	CLB Registers	CARRY8	F7 Muxes	F8 Muxes	CLB
BRAM	33	0	4	0	0	15
BRAM Input	0	0	2	0	0	2
Convolution	42715	9113	3808	0	0	10412
Lift	4015	1586	18	450	225	784
Controller	26469	125	400	0	0	8954
Multiplication Counter	44	70	0	0	0	47
Polynomial Register	602	9113	0	0	0	3905
Reduction	8400	13	0	0	0	1402
Ternary Register	5601	1404	1	0	0	2444
Full Implementation	87166	21424	5633	450	225	14408

Table 5.1: Hardware implementation hierarchical view results.

However, the implementation phase contains several optimizations that influence the hierarchical area profile by a significant amount, sometimes shifting a large amount of LUTs from one component to another. To give a more accurate overview of the percentage of area that is used by each component, Table 5.2 shows the synthesis results of each component separately so that each component can be analyzed and future improvements to high area usage components can be discussed.

Component Name	LUTs	Registers
BRAM	26	0
Convolution	50798	9113
Lift	3062	1515
Controller	260	113
Multiplication Counter	2589	93
Polynomial Register	4906	9113
Reduction	26946	9113
Ternary Register	702	1404

Table 5.2: Hardware synthesis individual component results.

Additionally, Table 5.3 shows the amount of clock cycles that a full encryption and decryption cycle for this implementation, including the effect of the input and output states on the encryption and decryption cycles respectively.

Function	Clock Cycles
Encryption	3038
Decryption	2876677

Table 5.3: Amount of clock cycles for the Encryption and Decryption function.

From Table 5.3 it can be seen that encryption can be performed fast, while the decryption takes a large amount of cycles (over 1000x slower than the encryption). This is mainly due to the final polynomial multiplication required in the decryption and this will be discussed further in section 5.2. It should be noted that the decryption time changes based on the input due to the final polynomial multiplication and the amount of cycles shown in Table 5.3 is the average amount of cycles. To get a better overview and give more contrast between the area usage described above and the amount of clock cycles, all functions used within the encryption and decryption cycles are analyzed separately below.

Overhead Results Analysis

The individual synthesis shown in Table 5.2 introduces logic elements, such as additional registers, that would not be there in the actual implementation because these registers would already be present in connected blocks. Nevertheless, when comparing the LUT usage of Table 5.1 and 5.2 it can be seen that a large amount of logic is moved to the controller block during implementation. The controller is just a state machine and Table 5.2 indicates that it is generally using up a small amount of area by itself, therefore a large part of this logic can be attributed to the reduction block and convolution block. Similarly, the ternary register is just a shift register, indicating that a large part of the post-implementation logic comes from the aforementioned blocks instead.

The convolution block is used for a large part of the encryption and decryption steps and has to perform multiple different polynomial multiplications, therefore a large amount of area usage is to be expected. The reduction block, however, outputs to multiple different other blocks and works in parallel. Due to this design decision, the area usage of the reduction block is higher than would be desired because of the large amount of parallel output multiplexing required alongside the parallel adders and module components. Section 5.4 will go over an optimization for this to reduce the amount of LUTs in this block.

Analysis of the amount of registers in the final implementation shows that this is roughly the expected amount of registers for this design:

- **Convolution:** q-bit (13) shift register that is n (701) wide = 9113.
- **Polynomial Register:** q-bit (13) shift register that is n (701) wide = 9113.
- **Ternary Register:** 2-bit shift register that is $n+1$ (702) wide = 1404
- **Lift:** 2-bit shift register that is $n+1$ (702) wide and three 11-bit registers for dot product calculation = 1437

This adds up to 21067 expected register, which leaves 357 registers for the multiplication counter, state machines, and state counters.

5.3. Performance Results

The performance and validity of the implementation can be evaluated by looking at the simulation results, as well as the block specific area usage results. Both the Lift and Convolution function perform a majority of the computations required for encryption and decryption, therefore their simulation will be analyzed to confirm whether they compute their output correctly and to give an overview of the signal changes between clock cycles.

5.3.1. Lift Function Results

Table 5.4 shows the clock cycle breakdown of the Lift function. Computing the dot product refers to the calculating of the first three coefficients of **a**, while the second step is the amount of clock cycles required to compute the remaining coefficients of the polynomial **a**. The output polynomial **b** is calculated in parallel and therefore takes one clock cycle to compute.

Function	Clock Cycles
Compute Dot Product	234
Calculate intermediate polynomial a	233
Calculate output polynomial b	1
Total	468

Table 5.4: Amount of clock cycles for the Lift function divided into the different steps performed in the lift algorithm.

The Lift function can be split up into separate segments when simulating to analyze the behaviour of the different steps. Figure 5.2 shows the waveform of the simulation while calculating the dot product to compute coefficients $a_0, a_1,$ and a_2.

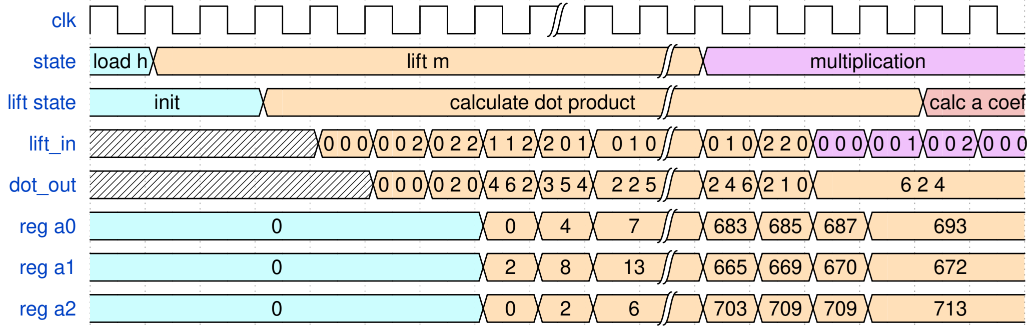


Figure 5.2: Waveform of the Lift function simulation during the computing of coefficients $a_0, a_1,$ and a_2.

This first computation uses the static vector shown in Table 4.2, where the first two cycles have a static output and the remaining cycles all have a periodic output. Keeping in mind that the first input has a padded zero, the expected output for the two static multiplications is:

$$
\begin{aligned}
\text{output}[0] &= \text{input}[1] \\
\text{output}[1] &= \text{input}[2] \\
\text{output}[2] &= \text{input}[0] \\
\text{output}[3] &= \text{input}[0] + 2 \cdot \text{input}[1] \\
\text{output}[4] &= \text{input}[1] + 2 \cdot \text{input}[2] \\
\text{output}[5] &= \text{input}[0] + \text{input}[2]
\end{aligned}
$$

While the periodic output will always be:

$$
\begin{aligned}
\text{output[i]} &= \text{input}[0] + 2 \cdot \text{input}[1] \\
\text{output[i+1]} &= \text{input}[1] + 2 \cdot \text{input}[2] \\
\text{output[i+2]} &= 2 \cdot \text{input}[0] + \text{input}[2]
\end{aligned}
$$

From Figure 5.2 it can be seen that the first input is $\{0,0,2\}$, corresponding to $\{0, m_0, m_1\}$, which should therefore result in the output $\{m_1, m_2, 0\}$ and this is equal to $\{0,2,0\}$. These are the first values that are stored in the registers for $a_0, a_1,$ and a_2 in the simulation as well. The second static input is $\{0,2,2\}$, corresponding to $\{m_2, m_3, m_4\}$, which should therefore result in the output $\{m_3 + 2 \cdot m_4, m_4 + 2 \cdot m_5, m_3 + m_5\}$, and this is equal to $\{4,6,2\}$. In the simulation this is the second output of the dot product function and it can be seen that this output is correctly added to the registers storing intermediate values of $a_0, a_1,$ and a_2. This same check can be performed for the periodic part of the dot product calculation, resulting in an expected output of $\{3,5,4\}$ which is the result shown in the simulation waveform as well.

From the waveform it can also be seen that the polynomial multiplication in the Convolution block is correctly started after performing the initial dot product computation because the Lift function no longer requires inputs from the BRAM after this step, allowing both to run in parallel.

After the periodic part of the dot product calculation the lift state changes to the calculating **a** coefficient state with values of $\{693, 672, 713\}$ in the registers for $a_0, a_1,$ and a_2 respectively. Although these coefficients of **a** are temporary values used in the computing of **m'**, it is possible to compare these values with the values in the software implementation [80] provided alongside the NTRU documentation and this implementation has the same values for $a_0, a_1,$ and a_2 at this point in the Lift function.

The waveform for the calculating **a** coefficients step of the Lift function can be seen in Figure 5.3

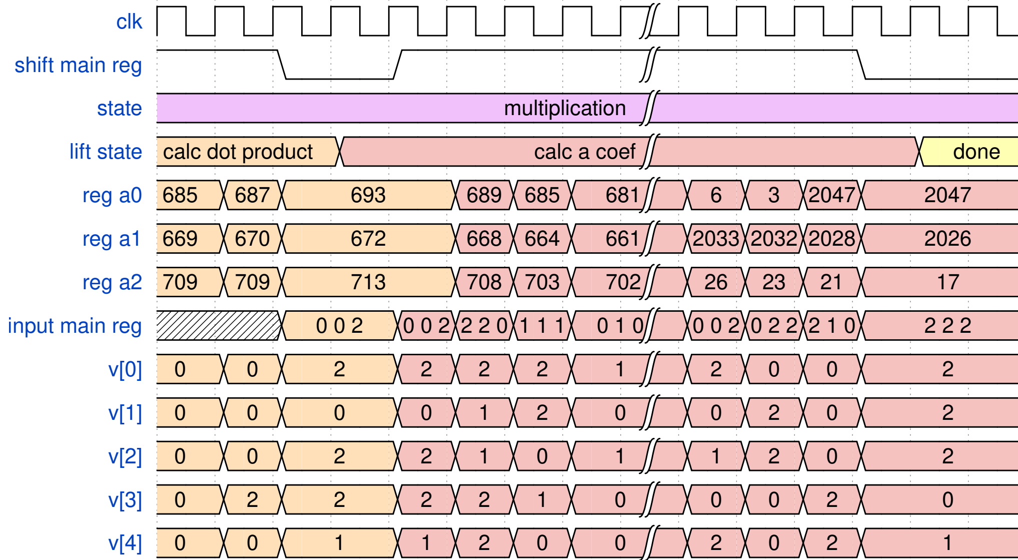


Figure 5.3: Waveform of the Lift function simulation during the computing of the remaining coefficients of **a**.

Since computing the remaining coefficients of **a** is done using Equation 4.4, there are always five different values of **v** required to compute three values of **a** at once. Due to the fact that the coefficient of **m** is not needed to compute the remaining coefficients of **a** and the fact that there is a padded zero in the first output register as well, the input v_0 to v_4 is defined the following:

$$\{v_0, v_1, v_2, v_3, v_4\} \equiv \{\text{reg}[699], \text{reg}[698], \text{reg}[697], \text{reg}[696], \text{reg}[695],\}$$

When the calculating **a** coefficients step starts a_0, a_1, and a_2 are already stored in separate registers, these go through a modulo 3 block and serve as the first input of the main lift registers resulting in an input of $\{693\%3, 672\%3, 713\%3\}$ which is equal to $\{0, 0, 2\}$ and from the waveform in Figure 5.3 it can be seen that this is the initial main register input before any form of register shifting starts. Additionally, the initial input $\{v_0, v_1, v_2, v_3, v_4\}$ is equal to $\{m_1, m_2, m_3, m_4, m_5\}$, indicating that the entirety of input **m** has been stored inside the main register while doing the dot product function.

The required computation each cycle can be defined the following:

$$\begin{aligned}
\text{new a[0] reg value} &= \text{old a[0] reg value} - (v_0 + v_1 + v3) \\
\text{new a[1] reg value} &= \text{old a[1] reg value} - (v_1 + v_2 + v_3) \\
\text{new a[2] reg value} &= \text{old a[2] reg value} - (v_2 + v_3 + v_4)
\end{aligned}$$

where the a[0] register will contain $a_3 \rightarrow a_6 \rightarrow a_9$ and so on. In the waveform this means that each cycle the value for the registers of a_0, a_1, and a_2 has to be replaced by the three next coefficients respectively and the old value in these registers will be the input for the main register (modulo 3) so that it is not lost. From Figure 5.3 it can be seen that the register of a_0 is replaced by a_3 with a value of 689, which is equal to $a_0 - (v_0 + v_1 + v_2) \equiv 693 - (2 + 0 + 2)$ and a similar operation happens for a_4 and a_5. At this point a_3, a_4, and a_5 become inputs for the main register resulting in $\{2, 2, 0\}$, which corresponds with the simulation result, and the main register is shifted by three spots to get the new values of $\{v_0, v_1, v_2, v_3, v_4\}$ that are required for the next three coefficients. It can also be seen that during the last few computations the coefficients start getting negative values and therefore overflow, but because a modulo 3 operation is performed before the coefficients are stored in the main register this does not influence the Lift function. Lastly, at the end of this step the output registers shown in the waveform are $\{2, 2, 2, 1, 0\}$ which corresponds to the values of $\{a_2, a_3, a_4, a_5, a_6$ indicating that the values of a_1 and a_2 might have been lost. However, it can be seen from the waveform that these two values, which are both 0 in the simulation, have been shifted forward into the two registers that were not used in the computation.

5.3.2. Convolution Function Results

Table 5.5 shows the clock cycle breakdown of the Convolution function. Since a large part of the logic is reused for each multiplication, with multiplex signals being different, the amount of clock cycles for each multiplication is listed separately.

Function	Clock Cycles
Multiplication in Encryption	704
1st Multiplication in Decryption	704
2nd Multiplication in Decryption	704
3rd Multiplication in Decryption	2871296

Table 5.5: Amount of clock cycles for the Convolution function, where each multiplication in the NTRU algorithm is listed separately.

The table shows that each multiplication takes an amount of cycles that is almost equal to the polynomial length but slightly higher due to the delay on the BRAM output and switching states, with the exception of the final multiplication taking 4078545% longer than the other multiplication. This highlights the main cause of the long decryption cycle and it should be noted that this multiplication length changes depending on the values of the input polynomials. As more coefficients get closer to the maximum value of q the amount of additions required will go up, therefore the average value of this multiplication would be $n \cdot (q/2)$ which is 2871296 cycles.

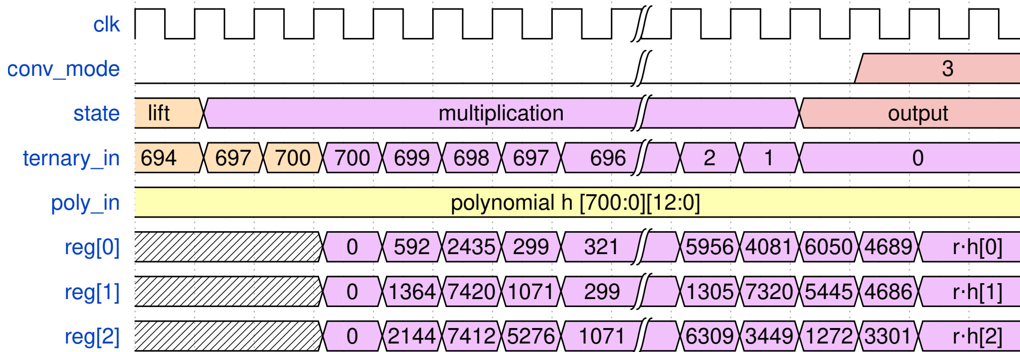


Figure 5.4: Waveform of the Convolution function simulation during the encryption multiplication $r \cdot h$.

Figure 5.4 shows the waveform of the simulation while performing the multiplication in encryption. Only the first three registers in the convolution block are shown in this figure, but these can be used to validate a large part of the multiplication. The waveform has been divided into different colors to indicate the different states and their respective inputs and outputs. Provided that the theory behind the convolution block is proven to be correct [70], since all the registers have identical logic in between, inspecting the intermediate values of one register should provide sufficient validation for all intermediate values. The intermediate values are propagated through the shift register and therefore one incorrect intermediate value would result in an incorrect final multiplication output. Therefore, by analyzing the intermediate values of one register and the final output polynomial, the validity of this multiplication can be validated.

To analyze the validity of the intermediate values of reg[1] the inputs to the function must be known:

- ternary polynomial **r** with coefficients $\{1,2,2,0,...,1,1,1,2\}$.
- polynomial **h** with coefficient $h[0] = 1364$, $h[1] = 2144$, and $h[700] = 592$.

Referring back to the convolution module for encryption schematic in Figure 4.5, it can be seen that the next value in a register is determined by the previous value of the register before it and an added multiplication between **h** and **r**. For reg[1] this means the value of $h[0]$ is always used to compute the next register value for reg[1]. The first ternary input is 1, therefore to get the value of reg[1]:

$$\text{reg}[1] = \text{reg}[0] + h[0] = 0 + 1364 = 1364$$

This is the first value that shows up in the waveform of Figure 5.4 as well. For the second input value where the ternary input is binary 2, which represents a value of -1 in the algorithm, the expected value for reg[1] is:

$$\text{reg}[1] = \text{reg}[0] - h[0] = 592 - 1364 = -772 \equiv 7420 \text{ in mod } q$$

Which is the 2nd value that shows up in the waveform for Figure 5.4. This process can be repeated for each step of the simulation. Additionally, because the expected final outcome is known during testing, it can be seen that at the end of the simulation the value of reg[1] is 4686, which is equal to the value of $r \cdot h[0]$. Similarly the value of reg[2] is equal to the value of $r \cdot h[1]$, indicating that an additional shift has to be performed. However, to avoid spending an additional clock cycle performing this shift, the output connection can be shifted as well, which is what was done in the implementation. Both the first and second polynomial multiplications in decryption have been tested and validated in this similar way, with the only difference being the mode signal having a different value to select the right logic in between registers. However, the third polynomial multiplication in decryption is performed different and therefore the waveform for this multiplication is shown below in Figure 5.5

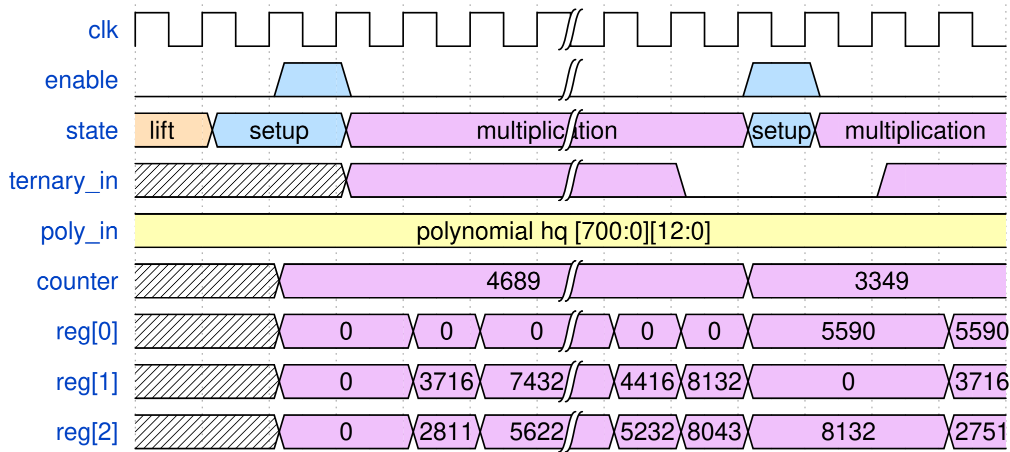


Figure 5.5: Waveform of the Convolution function simulation during the third decryption multiplication $(c - m') \cdot h_q$.

The third polynomial multiplication in decryption uses the multiplication counter block to determine how often a ternary 1 has to be given as input for the Convolution function. Due to this, additional signals have been added to the waveform, namely the enable signal and the counter signal. Additionally, the ternary value is only ever 1 or 0 in this multiplication and shown accordingly. The counter block receives coefficients of **c** from the BRAM and coefficients of **m'** from the lift block to compute $\mathbf{c} \cdot \mathbf{m}'$ and these are forwarded to the Convolution function when an enable signal is given. At the start of the third polynomial multiplication the setup state is two clock cycles, one cycle to reset all registers in the Convolution function and one cycle to retrieve the counter value by means of this enable signal. It can be seen that the first counter value is 4689 which is equal to $c[700] - m'[700] \equiv 4690 - 1$ and this means that the Convolution function will receive 4689 ternary values of 1. Similarly to the other multiplication, a ternary value of 1 in the shown register indices, corresponds to:

$$\text{reg}[0] = \text{reg}[700] + h_q[700] = 0 + 0 = 0$$
$$\text{reg}[1] = \text{reg}[0] + h_q[0] = 0 + 3716 = 3716$$
$$\text{reg}[2] = \text{reg}[1] + h_q[1] = 0 + 2811 = 2811$$

Because the Convolution function is only receiving ternary values of 1 this process keeps repeating, effectively adding the values of h_q, as can be seen in Figure 5.5. At the end of one multiplication, the setup state starts again and the enable signal is used to set the next coefficient that needs to be multiplied, repeating the process. Unlike the other multiplications, the shift at the end has to be performed and it can be seen that this also happens in the setup state, where reg[1] gets the shifted value of reg[0].

5.3.3. Additional Performance Results

The input and output cost of both the encryption and decryption can be seen in Table 5.6.

Function	Clock Cycles
Input Encryption	1402
Output Encryption	701
Input Decryption	701
Output Decryption	1402

Table 5.6: Amount of clock cycles for the different input and output sequences for both the encryption and decryption.

Since the encryption requires two polynomials (**r** and **m**) as input, the amount of cycles is twice as much as the decryption which only requires the ciphertext **c** as an input. The opposite relation holds for the output and therefore the amount of cycles for input and output combined is the same for both encryption and decryption (2103).

Block (Function)	Clock Cycles
Reduction (All Reductions)	1
Ternary Register (Load m)	1
Polynomial Register (Load h)	701
Polynomial Register (Load f)	701
Polynomial Register (Load a)	1
Polynomial Register (Load h_q)	701

Table 5.7: Amount of clock cycles for all remaining operations required for the encryption and decryption steps.

All remaining functions that are used by encryption and/or decryption are listed in Table 5.7. Since both the reduction and loading of the ternary register are performed in parallel, both take a single clock cycle. In the case of the polynomial register, input **h**, **f**, and $\mathbf{h_q}$ are loaded from the BRAM at one coefficient per cycle and therefore takes 701 cycles, which is equal to the polynomial length n. Input **a** is loaded in parallel from the reduction block and therefore takes one cycle.

5.4. Implementation Discussion

Looking at the results, there are several improvements that can be made to the implementation to either reduce the amount of area or increase the speed at which decryption happens.

Early on in the project the design decision was made to perform the last step of the lift function parallel, the reasoning for this is that the input of this step could be given in parallel and the output could be given in parallel as well. Additionally, the parallel implementation was appealing due to the simplicity of the design as well. However, this parallel design has a large impact on the area usage of the overall implementation due to the fact that it requires roughly $2n$ adders and n modulo 3 components, as well as many blocks that perform a mapping. When designing the output state of the design several parallel outputs had to be serialized and this led to adjustments in the design where a parallel output of the lift function was no longer required. Due to this there is room to improve the last step of the lift function by only using five inputs and outputs and using the shift registers that are already in the design to serialize the output. This improvement would drastically reduce the amount of logic in the lift function and therefore reduce the area usage of the design. This would not affect the performance of the encryption step because the encryption output **c** is dependant on the output of both the Lift function and the Convolution function. Even when both of these are performed in parallel, the Convolution function takes 1402 ($2n$) cycles to load the inputs and perform the multiplication. Adjusting the parallel part of the Lift function to be serial would at most add 701 (n) cycles and still end up taking a shorter amount of cycles than what the Convolution function needs.

It would, however, affect the performance of the decryption step, but the decryption is largely bottlenecked by the final polynomial multiplication, so much so that this increase in cycles has a < 1% impact on the overall performance.

In the same vein, the reduction block (which is only used in the decryption) suffers from similar large area usage that could be serialized at the cost of a negligible performance hit to the decryption, reducing the amount of area that this block uses from n adders and modulo 3 components to 5 adders and modulo 3 components.

As mentioned before, the third polynomial multiplication is the largest bottleneck of the decryption state. However, there are several ways to increase the speed of this polynomial multiplication. In this implementation

the choice has been made to go for the lowest area option, but trade-offs can be made, increasing the area of the convolution block to reduce total execution time. Qin et al.[71] have shown that this multiplication can be sped up by using Booth-Code, which calculates two bits of a number at once instead of one, effectively doubling the speed at which this multiplication is performed. However, a trade-off between the amount of DSP units in the design can also be made. Given a single DSP, each coefficient multiplication can be reduced to take n clock cycles, resulting in a multiplication with time complexity $\mathcal{O}(N^2)$. By adding more DSP units and increasing the area usage of the implementation this multiplication, this multiplication can be sped up much more, allowing hardware that has additional area to reduce the execution time of this multiplication as they see fit. For small devices that are not affected by the longer execution time this would not be needed, however, the convolution block can be adjusted in this way to accommodate larger devices that require a faster decryption step.

Software comparison

In the NTRU package there are two software implementations provided, the reference implementation using C instructions and an optimized AVX2 implementation, using AVX2 vector instructions. Both of these software implementations have been benchmarked on one core of an Intel Core i7-4770K (Haswell) and the amount of cycles taken for both encryption and decryption for these implementations can be seen in Table 5.8.

	Software implementation - C	Software implementation - AVX2	Hardware implementation
Encryption	1069326 cycles	50441 cycles	3038 cycles
Decryption	3113303 cycles	62267 cycles	2876677 cycles

Table 5.8: Amount of clock cycles for the encryption and decryption for both software implementations in the NTRU package and this hardware implementation.

The software benchmark results shown in Table 5.8 were performed with 32 KiB L1 data cache, 256 KiB L2 cache and 3072 KiB L3 cache. Furthermore, it has 16GiB of RAM, running at 1066 MHz. On the other hand, the hardware implementation using the Kintex UltraScale+ KCU116 Evaluation Platform can reach clock speeds up to 945 MHz, while testing has been performed using a 250 MHz clock. At the maximum hardware clock speed the clock speeds are similar and cycles can be compared, however, hardware using a lower clock frequency, which is more realistic for IoT devices, will compare worse to the software than the results in Table 5.8. Additionally, software running on a device with a faster clock will have a similar impact on the comparison results.

From Table 5.8 it can be seen that the encryption in hardware is much faster than both software implementation resulting in a speedup of 352 and 16 when compared to the C implementation and AVX2 implementation respectively. When it comes to decryption the hardware implementation is 0.95 times the speed of the C implementation and 0.02 times the speed of the AVX2 implementation, mainly due to the impact of the third polynomial multiplication required in the decryption step. It should be noted that without this multiplication, the rest of the decryption process takes 5381 clock cycles in hardware. If any of the improvements to the third multiplication discussed above were implemented, or additional area usage was of no concern for a specific hardware implementation, this multiplication could be reduced to a similar amount of clock cycles as the other multiplications, being 704 clock cycles. More realistically however, adding 50 DSP units to the design could reduce this multiplication to take about 10000 clock cycles ($n \cdot \frac{n}{50}$) resulting in 15381 cycles for decryption. Additionally, the use of DSP units removes the variable amount of cycles based on the input, resulting in a more stable implementation.

5.5. Security Analysis

Even though hardware implementations can provide a significant speedup compared to software, the hardware logic can introduce new security vulnerabilities by being susceptible to side-channel analysis. A side-channel attack is can be achieved by analyzing physical parameters of the chip, such as execution time, temperature, supply current, and electromagnetic emission. If this analysis can be used to retrieve secret information from a chip, such as the private key in the NTRU algorithm, the security of the algorithm is compromised which means that the hardware implementation can no longer be used for any forms of secret information exchange.

An example of the impact of these side-channel attacks is the attack on NTRU performed by Amund Askeland and Sondre Rønjom [81]. In this attack a small component of the algorithm, the modulo 3 function, is shown to leak information about the private key polynomial **f** based on a difference in Hamming weight

that is caused by the if-else statement at the end of the modulo 3 function. Using this vulnerability they were able to retrieve 75% of the private key polynomial **f**. Even though this vulnerability is unique to the software implementation due to the different structure of if-else statements, it serves as an example that secret information can leak from unexpected functions and locations in the implementation.

Even though there are several methods to reduce the effects of side-channel analysis, such as blinding techniques to prevent leakage [82], or masking sensitive information [83], identifying the locations where information could potentially leak is required to perform any kind of countermeasure.

The NTRU algorithm has secret information in the form of $\mathbf{f}$, $\mathbf{f_p}$, $\mathbf{h_q}$, and **s**, with the combination of these four polynomials making up the entire private key. This can be considered an advantage because leaking the private key would require an adversary to obtain leaked information of all four of these polynomials, of which three are used in the decryption functions ($\mathbf{f}$, $\mathbf{f_p}$, and $\mathbf{h_q}$) that have been implemented in this hardware. However, logic where all three of these polynomials are used is therefore the most at risk when it comes to side-channel analysis, since the leaked information in this logic could be used to retrieve the private key. In the hardware implementation there are three blocks where all three of the aforementioned polynomials are used:

- The BRAM contains a stored version of each polynomial and this includes the full private key.
- The Polynomial register block is used to load polynomial $\mathbf{f}$ and $\mathbf{h_q}$.
- The Convolution block performs a multiplication using polynomial $\mathbf{f}$, $\mathbf{f_p}$, and $\mathbf{h_q}$.

Since the private key is in the BRAM at all times, any form of attack on the BRAM could reveal the private key. The current implementation uses a single-port BRAM to store each polynomial, making it both simpler to use and simpler to perform an attack on by an adversary. One countermeasure that could be implemented to make an attack on the BRAM more difficult would be to perform masking using some form of data scrambling [84] so that the polynomials are no longer stored as plain text and therefore less likely to leak information. Additionally, refreshing the private key after a certain amount of time would reduce the impact of a side-channel attack on the BRAM, as well as generating a new private key on boot each time to reduce the impact of cold boot attacks. This is a type of side channel attack that takes advantage of data remanence, which is the residual representation of digital data that remains even after attempts have been made to remove or erase the data.

The security of the polynomial register block and the convolution block are connected, since the polynomial register block serves as an input for the polynomial multiplications performed in the convolution block. For most of the operations performed in these blocks the operations are independent of the input, however, during the multiplication the combinatorial logic between registers will respond differently based on the ternary polynomial input. Even though it is difficult to determine whether information is being leaked due to this, blinding can be implemented here by always computing all three potential outputs independent of the ternary input, reducing the amount of potential leakage in the convolution block.

6 Conclusion

This chapter provides a summary of the contributions and achievements of this thesis while also highlighting potential future research in the field. Section 6.1 gives a summary of each chapter and goes over their contributions. Section 6.2 discusses several future work directions for NTRU hardware implementations.

6.1. Summary

Chapter 1 of this thesis introduced the need for quantum resistant cryptoschemes due to the increasing threat of quantum computing. The current heavily relied upon public-key encryption and digitial signature algorithms can be brute forced by large enough quantum computers, endangering the security of the cyberspace. This gave motivation for the topic of the thesis, a hardware implementation of NTRU, which is one of the current post-quantum cryptoscheme candidates in the NIST Post-Quantum Cryptography Standardization effort.

Chapter 2 discussed the history of the NIST Post-Quantum Cryptography Standardization effort and the candidates that are part of the current round of the competition. The chapter discussed the different types of KEM schemes and how they differ from one another. Since NTRU is a lattice-based cryptoscheme it discussed the different complex problems that the numerous lattice-based cryptoschemes are based around, connecting each problem back to the base shortest vector problem. Outside of discussing what problems these cryptoschemes are based on, this chapter also discussed the strengths of each lattice-based cryptoscheme and why so many candidates remain in this category.

Chapter 3 of this thesis presented the NTRU algorithm, discussing the different parameter sets that can be used and what effects these parameters have on the implementation of the algorithm. It divided the algorithm up into blocks to give a better overview of both the encryption and decryption function and what type of computations are required to implement these. The chapter also discussed what types of polynomial multiplication methods are available as this is a majority of the computational work required in the NTRU cryptoscheme.

Chapter 4 went over the full hardware implementation by going over the states of the system in order and building up to the final system block diagram. All blocks that make up the overall system were discussed individually to indicate what design decisions had been made and what the inner workings of each component were.

Chapter 5 presented the results of the hardware implementation, showing the area usage of each component to provide an analysis of the overall implementation. It discussed the LUT and register usage and highlighted blocks that had a higher area cost than desired. After analysis this chapter discussed several improvements that can be made to the implementation to reduce both the amount of area used and execution time. Finally, it demonstrated that a full encryption and decryption routine can be performed using this implementation and provided the amount of time it takes for both encryption and decryption.

6.2. Future Work

There are several recommendations that would enhance the topics that have been covered in this thesis either by performing additional background research or by adding to the implementation. The recommendations are listed below:

- Further analysis of the different NIST PQC candidates, including the signature algorithms. This thesis mainly focused on the lattice-based KEM cryptoschemes that are part of the current round of the NIST PQC standardization effort, however, both the code-based and isogeny-based KEM cryptoschemes show much promise as well. Additionally, the signature schemes were not discussed in this thesis and further background research into these schemes is recommended.

- Section 5.4 went over the improvements that can be made to the polynomial multiplication step of the algorithm, specifically the final multiplication. Further investigation can be done on these improvements regarding the adding of DSP units to the implementation and analyzing the speed and area trade-off based on the number of DSP units that are added.
- As discussed in section 5.4, two components of the current implementation have been designed to work in parallel, while serializing these components would likely improve the overall implementation. Unfortunately due to timing constraints this was not implemented during this project and therefore looking into the total improvement of this change is recommended.
- Key generation is not part of the current implementation due to the fact that it requires specialized hardware to perform polynomial division. However, to have a full implementation of the NTRU algorithm research can be done into implementing these key generation steps in hardware.
- A hardware implementation of the two sample functions that are used at the start of the key generation and encryption step, along with the encapsulation and decapsulation steps, are not part of the current implementation. Additional time and research is required to implement this, as well as looking into different sample functions that may be better suited for hardware
- Due to the fact that hardware implementation are vulnerable to a different set of attacks than software implementations, such as side-channel analysis, analyzing the NTRU implementation and looking for vulnerabilities is required before any form of security can be guaranteed.

Bibliography

[1] P. W. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,” SIAM Journal on Computing, vol. 26, no. 5, pp. 1484–1509, oct 1997. [Online]. Available: https://doi.org/10.1137%2Fs0097539795293172

[2] M. Capra, R. Peloso, G. Masera, M. Ruo Roch, and M. Martina, “Edge computing: A survey on the hardware requirements in the internet of things world,” Future Internet, vol. 11, p. 100, 04 2019.

[3] “International conference on post-quantum cryptography.” [Online]. Available: https://2022.pqcrypto.org/index.html

[4] “NIST post-quantum cryptography standardization.” [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization

[5] H. Riel, “Quantum computing technology,” in 2021 IEEE International Electron Devices Meeting (IEDM), 2021, pp. 1.3.1–1.3.7.

[6] Hardware encryption market revenue & trend forecast report, 2019-2026. [Online]. Available: https://www.marketdecipher.com/report/hardware-encryption-market

[7] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in 40th IEEE Symposium on Security and Privacy (S&P’19), 2019.

[8] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown: Reading kernel memory from user space,” in 27th USENIX Security Symposium (USENIX Security 18), 2018.

[9] H. Nejatollahi, R. Cammarota, F. Regazzoni, I. Banerjee, and N. Dutt, “Software and hardware implementation of lattice-based cryptography schemes,” Tech. Rep., 11 2017.

[10] S. Wiesner, “Conjugate coding,” SIGACT News, vol. 15, no. 1, p. 78–88, jan 1983. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/1008908.1008920

[11] C. H. Bennett, F. Bessette, G. Brassard, L. Salvail, and J. Smolin, “Experimental quantum cryptography,” J. Cryptol., vol. 5, no. 1, p. 3–28, jan 1992.

[12] A. K. Ekert, “Quantum cryptography based on bell’s theorem,” Phys. Rev. Lett., vol. 67, pp. 661–663, Aug 1991. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.67.661

[13] C. H. Bennett, G. Brassard, S. Popescu, B. Schumacher, J. A. Smolin, and W. K. Wootters, “Purification of noisy entanglement and faithful teleportation via noisy channels,” Physical Review Letters, vol. 76, no. 5, pp. 722–725, jan 1996. [Online]. Available: https://doi.org/10.1103%2Fphysrevlett.76.722

[14] D. Deutsch, A. Ekert, R. Jozsa, C. Macchiavello, S. Popescu, and A. Sanpera, “Quantum privacy amplification and the security of quantum cryptography over noisy channels,” Physical Review Letters, vol. 77, no. 13, pp. 2818–2821, sep 1996. [Online]. Available: https://doi.org/10.1103%2Fphysrevlett.77.2818

[15] A. Ekert, J. Rarity, P. Tapster, and G. Palma, “Practical quantum cryptography based on two-photon interferometry,” Physical review letters, vol. 69, pp. 1293–1295, 09 1992.

[16] M. Ajtai, “Generating hard instances of lattice problems (extended abstract),” in Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing, ser. STOC ’96. New York, NY, USA: Association for Computing Machinery, 1996, p. 99–108. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/237814.237838

[17] D. Micciancio, Shortest Vector Problem. New York, NY: Springer New York, 2016, pp. 1974–1977. [Online]. Available: https://doi.org/10.1007/978-1-4939-2864-4_374

[18] D. Micciancio and O. Regev, Lattice-based Cryptography. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 147–191. [Online]. Available: https://doi.org/10.1007/978-3-540-88702-7_5

[19] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A new high speed public key cryptosystem," draft from CRYPTO '96 rump session, 1996. [Online]. Available: https://web.securityinnovation.com/hubfs/files/ntru-orig.pdf

[20] R. J. McEliece, "A Public-Key Cryptosystem Based On Algebraic Coding Theory," Deep Space Network Progress Report, vol. 44, pp. 114–116, Jan. 1978.

[21] O. Goldreich, S. Goldwasser, and S. Halevi, "Public-key cryptosystems from lattice reduction problems," in Advances in Cryptology — CRYPTO '97, B. S. Kaliski, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 112–131.

[22] P. Nguyen, "Cryptanalysis of the goldreich-goldwasser-halevi cryptosystem from crypto '97," in Advances in Cryptology — CRYPTO' 99, M. Wiener, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 288–304.

[23] P. Q. Nguyen and O. Regev, "Learning a parallelepiped: Cryptanalysis of ggh and ntru signatures," in Advances in Cryptology - EUROCRYPT 2006, S. Vaudenay, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 271–288.

[24] D. Micciancio, "Generalized compact knapsacks, cyclic lattices, and efficient one-way functions from worst-case complexity assumptions," Cryptology ePrint Archive, Paper 2004/286, 2004, https://eprint.iacr.org/2004/286. [Online]. Available: https://eprint.iacr.org/2004/286

[25] O. Regev, "New lattice-based cryptographic constructions," J. ACM, vol. 51, no. 6, p. 899–942, nov 2004. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/1039488.1039490

[26] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS – kyber: a CCA-secure module-lattice-based KEM," Cryptology ePrint Archive, Paper 2017/634, 2017. [Online]. Available: https://eprint.iacr.org/2017/634

[27] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in Algorithmic Number Theory, J. P. Buhler, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 267–288.

[28] "Ntru prime," 2017, submission to NIST post-quantum call for proposals.

[29] J.-P. D'Anvers, A. Karmakar, S. Sinha Roy, and F. Vercauteren, "Saber: Module-LWR based key exchange, cpa-secure encryption and CCA-secure KEM," in Progress in Cryptology – AFRICACRYPT 2018, A. Joux, A. Nitaj, and T. Rachidi, Eds. Cham: Springer International Publishing, 2018, pp. 282–305.

[30] 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," Cryptology ePrint Archive, Paper 2016/659, 2016. [Online]. Available: https://eprint.iacr.org/2016/659

[31] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, H. Zalcman, Adamand Neven, and J. M. Martinis, "Quantum supremacy using a programmable superconducting processor," Nature, vol. 574, no. 7779, pp. 505–510, Oct 2019. [Online]. Available: https://doi.org/10.1038/s41586-019-1666-5

[32] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," Commun. ACM, vol. 21, no. 2, p. 120–126, feb 1978. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/359340.359342

[33] N. Koblitz, "Elliptic curve cryptosystems," Mathematics of Computation, vol. 48, pp. 203–209, 1987.

[34] V. Miller, "Use of elliptic curves in cryptography." 01 1985, pp. 417–426.

[35] W. Diffie and M. Hellman, "New directions in cryptography," IEEE Transactions on Information Theory, vol. 22, no. 6, pp. 644–654, 1976.

[36] Moody, Dustin. The beginning of the end: The first NIST PQC standards. NIST. [Online]. Available: https://csrc.nist.gov/csrc/media/Presentations/2022/the-beginning-of-the-end-the-first-nist-pqc-standa/images-media/pkc2022-march2022-moody.pdf

[37] H. Singh, "Code based cryptography: Classic mceliece," 2019. [Online]. Available: https://arxiv.org/abs/1907.12754

[38] N. Aragon, P. S. L. M. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Guneysu, C. Aguilar Melchor, R. Misoczki, E. Persichetti, N. Sendrier, J.-P. Tillich, and G. Zémor, "BIKE: Bit Flipping Key Encapsulation," Dec. 2017, submission to the NIST post quantum standardization process. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01671903

[39] C. Aguilar, O. Blazy, J.-C. Deneuville, P. Gaborit, and G. Zémor, "Efficient encryption from random quasi-cyclic codes," IEEE Transactions on Information Theory, vol. PP, 12 2016.

[40] N. Sendrier, Niederreiter Encryption Scheme. Boston, MA: Springer US, 2011, pp. 842–843. [Online]. Available: https://doi.org/10.1007/978-1-4419-5906-5_385

[41] A. Valentijn, "Goppa codes and their use in the mceliece cryptosystems," 2015, Syracuse University Honors Program Capstone Projects. [Online]. Available: https://surface.syr.edu/honors_capstone/845/

[42] S. Ouzan and Y. Be'ery, "Moderate-density parity-check codes," 2009. [Online]. Available: https://arxiv.org/abs/0911.3262

[43] I. Reed, "A class of multiple-error-correcting codes and the decoding scheme," Transactions of the IRE Professional Group on Information Theory, vol. 4, no. 4, pp. 38–49, 1954.

[44] D. E. Muller, "Application of boolean algebra to switching circuit design and to error detection," Transactions of the I.R.E. Professional Group on Electronic Computers, vol. EC-3, no. 3, pp. 6–12, 1954.

[45] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," Journal of The Society for Industrial and Applied Mathematics, vol. 8, pp. 300–304, 1960.

[46] E. Prange, "The use of information sets in decoding cyclic codes," IRE Transactions on Information Theory, vol. 8, no. 5, pp. 5–9, 1962.

[47] D. Jao and L. Feo, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies post-quantum cryptography," vol. 2011, 11 2011, pp. 19–34.

[48] D. Jao, R. Azarderakhsh, M. Campagna, C. Costello, L. De Feo, B. Hess, A. Jalali, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, J. Renes, V. Soukharev, and D. Urbanik, "SIKE: Supersingular isogeny key encapsulation," 2022. [Online]. Available: https://sike.org/

[49] K. T. W. Weierstrass, "Formeln und lehrsätze zum gebrauche der elliptischen functionen; nach vorlesungen und aufzeichnungen des herrn k. weierstrass, bearb. und hrsg. von h.a. schwarz," 1893.

[50] S. Singh, "IJARCCE an efficient and secure protocol for ensuring data storage security in cloud computing using ECC," 01 2016.

[51] Y. Wang and A. Faz-Hernández. Hertzbleed explained. [Online]. Available: https://blog.cloudflare.com/hertzbleed-explained/

[52] C. Costello, "The case for SIKE: A decade of the supersingular isogeny problem," Cryptology ePrint Archive, Paper 2021/543, 2021, https://eprint.iacr.org/2021/543. [Online]. Available: https://eprint.iacr.org/2021/543

[53] M. Ajtai, "The shortest vector problem in l2 is np-hard for randomized reductions (extended abstract)," in Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, ser. STOC '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 10–19. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/276698.276705

[54] G. Falcao, F. Cabeleira, A. Mariano, and L. Paulo Santos, "Heterogeneous implementation of a voronoi cell-based svp solver," IEEE Access, vol. 7, pp. 127 012–127 023, 2019.

[55] Y.-L. Chuang, C.-I. Fan, and Y.-F. Tseng, "An efficient algorithm for the shortest vector problem," IEEE Access, vol. 6, pp. 61 478–61 487, 2018.

[56] C. Peikert, "Lattice-based cryptography: Short integer solution (SIS) and learning with errors (LWE)," 2020. [Online]. Available: https://web.eecs.umich.edu/~cpeikert/pubs/slides-abit2.pdf

[57] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," J. ACM, vol. 60, no. 6, nov 2013. [Online]. Available: https://doi.org/10.1145/2535925

[58] A. Langlois and D. Stehle, "Worst-case to average-case reductions for module lattices," Cryptology ePrint Archive, Paper 2012/090, 2012, https://eprint.iacr.org/2012/090. [Online]. Available: https://eprint.iacr.org/2012/090

[59] (2022) PQC standardization process: Third round candidate announcement. [Online]. Available: https://csrc.nist.rip/News/2020/pqc-third-round-candidate-announcement

[60] (2020) Securing the post-quantum world. [Online]. Available: https://blog.cloudflare.com/securing-the-post-quantum-world

[61] (2020) Round 2 post-quantum tls is now supported in AWS KMS. [Online]. Available: https://aws.amazon.com/blogs/security/round-2-post-quantum-tls-is-now-supported-in-aws-kms/

[62] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, J. Kelsey, Y. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone, "Status report on the second round of the NIST post-quantum cryptography standardization process," 2020. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf

[63] C. for Efficient Embedded Security, "Efficient embedded security standard (eess) #1," http://www.ceesstandards.org. [Online]. Available: https://cir.nii.ac.jp/crid/1574231875396165504

[64] N. Howgrave-Graham, J. H. Silverman, and W. Whyte, "Choosing parameter sets for ntruencrypt with naep and sves-3," in Topics in Cryptology – CT-RSA 2005, A. Menezes, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 118–135.

[65] "Ieee standard specification for public key cryptographic techniques based on hard problems over lattices," IEEE Std 1363.1-2008, pp. 1–81, 2009.

[66] A. Hülsing, J. Rijneveld, J. Schanck, and P. Schwabe, "High-speed key encapsulation from NTRU," in Cryptographic Hardware and Embedded Systems – CHES 2017, W. Fischer and N. Homma, Eds. Cham: Springer International Publishing, 2017, pp. 232–252.

[67] T. Saito, K. Xagawa, and T. Yamakawa, "Tightly-secure key-encapsulation mechanism in the quantum random oracle model," in Advances in Cryptology – EUROCRYPT 2018, J. B. Nielsen and V. Rijmen, Eds. Cham: Springer International Publishing, 2018, pp. 520–551.

[68] C. Chen, O. Danba, J. HoffStein, A. Hülsing, J. Rijneveld, J. M. Schanck, T. Saito, P. Schwabe, W. Whyte, K. Xagawa, T. Yamakawa, and Z. Zhang, "NTRU algorithm specifcations and supporting documentation," 2020.

[69] H. Krawczyk, K. G. Paterson, and H. Wee, "On the security of the tls protocol: A systematic analysis," in Advances in Cryptology – CRYPTO 2013, R. Canetti and J. A. Garay, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 429–448.

[70] K. Braun, T. Fritzmann, G. Maringer, T. Schamberger, and J. Sepúlveda, "Secure and compact full NTRU hardware implementation," in 2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), 2018, pp. 89–94.

[71] Z. Qin, R. Tong, X. Wu, G. Bai, L. Wu, and L. Su, "A compact full hardware implementation of PQC algorithm NTRU," in 2021 International Conference on Communications, Information System and Computer Engineering (CISCE), 2021, pp. 792–797.

[72] M. Wera, "A compact HW-SW codesign of NTRU KEM," 2020.

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

[74] S. A. Cook and S. O. Aanderaa, "On the minimum computation time of functions," Transactions of the American Mathematical Society, vol. 142, pp. 291–314, 1969.

[75] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," Soviet Physics Doklady, vol. 7, p. 595, 12 1962.

[76] M.-J. O. Saarinen, "Hila5: On reliability, reconciliation, and error correction for ring-lwe encryption," Cryptology ePrint Archive, Paper 2017/424, 2017. [Online]. Available: https://eprint.iacr.org/2017/424

[77] B. Liu and H. Wu, "Efficient architecture and implementation for ntruencrypt system," in 2015 IEEE 58th International Midwest Symposium on Circuits and Systems (MWSCAS), 2015, pp. 1–4.

[78] Xilinx, "Vivado ML." [Online]. Available: https://www.xilinx.com/about/blogs/adaptable-advantage-blog/2022/Vivado-ML-2022-1-Now-Supports-Versal-Premium-Devices.html

[79] (2022) Xilinx kintex ultrascale+ fpga kcu116 evaluation kit. [Online]. Available: https://nl.mouser.com/new/xilinx/xilinx-kintex-ultrascale-kcu116-eval-kit/

[80] (2020) NTRU round 3 NIST submission package. [Online]. Available: https://ntru.org/release/NIST-PQ-Submission-NTRU-20201016.tar.gz

[81] A. Askeland and S. Rønjom, "A side-channel assisted attack on ntru," Cryptology ePrint Archive, Paper 2021/790, 2021, https://eprint.iacr.org/2021/790. [Online]. Available: https://eprint.iacr.org/2021/790

[82] H. Kim, "Thwarting side-channel analysis against rsa cryptosystems with additive blinding," Information Sciences, vol. 412-413, pp. 36–49, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025517307478

[83] E. Prouff and M. Rivain, "Masking against side-channel attacks: A formal security proof," in Advances in Cryptology – EUROCRYPT 2013, T. Johansson and P. Q. Nguyen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 142–159.

[84] M. Neagu and L. Miclea, "Data scrambling in memories: A security measure," 05 2014, pp. 1–6.